Preliminary Mesa System Documentation

by Charles M. Geschke, Charles Irby, Richard K. Johnsson, Edwin H. Satterthwaite and John D. Wick

December 1, 1976

This collection of documentation describes the initial release of the Mesa programming system, Mesa library packages, and operational procedures for the Alto. It is intended as a preliminary effort, for use by experienced systems programmers operating in the Parc and SDD/Palo Alto environment. Substantial evolution of both the system and this documentation should be anticipated.

The Mesa, language, compiler and programming system are the product of a long-standing research project at Parc, in which Chuck Geschke, Butler Lampson, Jim Mitchell, and Ed Satterthwaite have been the main participants. The current compiler was written by Geschke and Satterthwaite. They also wrote the debugger and support software needed to run Mesa programs on the Alto, in collaboration with Jim Mitchell and with Charles Irby, Richard Johnsson and John Wick of SDD/Palo Alto. Compiler testing has been done by Jim Frandeen of SDD/Palo Alto.

XEROX

PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

SYSTEM DEVELOPMENT DIVISION
3406 Hillview Avenue / Palo Alto / California 94304

This document is for internal Xerox use only.

PREFACE

December 1, 1976

<MESA-DOC>MESASYSTEM

You are reading the preliminary Mesa system documentation for the Alto. Both this documentation and the objects it describes will be changing rather rapidly over the next year. The Maxc directory <MESA-DOC> will contain up-to-date versions of the various sections of this document as well as the complete document. Each section is stamped with its release date. It is the reader's responsibility to check that his hardcopy version is as current as the time-stamp in the corresponding file stored on <MESA-DOC>. To the best of their ability, the authors will take care to insure that the documentation and the system facilities which it documents are kept in synchrony.

All suggestions as to the form, correctness, and understandability of this material should be funneled through Chuck Geschke who will assume the responsibility for coordinating modifications and extensions. All of us involved in the development of Mesa welcome feedback and suggestions on both the language and the system environment. The earlier such suggestions are offered in the development cycle the greater the likelihood of their having a significant impact on the resulting system.

This documentation is divided into four parts. Section 1 contains the descriptions of various system facilities (debugger, library packages, system functions, etc). Section 2 describes where to find (on Maxc) the Mesa source and object code referred to in Section 1 as well as the documentation. Section 3 is a a very brief how-to-do-it description for getting a full Mesa system onto your Alto disk and some directions for running the Alto/Mesa compiler. Section 4 defines the format of Mesa "image" files and describes the procedure for producing them. Good luck!

SECTION 1: SYSTEM FACILITIES

Introduction

This section is broken into several subsections each of which describes a more or less logically disjoint subset of system facilities. Each component will be changing -- some more rapidly than others. The standard Mesa environment, which you may establish on your Alto disk by following the directions in Section 3, contains all of the facilities described in the initial release of this document -- binder, debugger, display package, string package, streams package, streamio package, segmentation (and file) machinery, and storage-allocation package. As the Mesa environment evolves, more packages and expanded facilities will emerge. As these extensions arrive, you will be notified (by message) on Maxc.

Each subsection is time-stamped and a pointer to the corresponding .BRAVO (and .EARS) file on <MESA-DOC> is printed at the beginning of the document. If you are uncertain about the accuracy of your copy of the particular subsection, you should check the time-stamp in the respective file on <MESA-DOC>. The appropriate DEFINITIONS modules, which user programs will want to reference, are named in the various subsections. This entire document appears in the <MESA-DOC> directory as MESASYSTEMDOCUMENT.EARS.

These are the subsections of Section 1 in alphabetical order:

Alto/Mesa Binding Facilities

March 1, 1976

<MESA-DOC>BIND

The facilities described below are used for binding a modules's external references (to PROCEDUREs, SIGNALs, and ERRORs) after it has been loaded. The default binding structure established when modules are loaded is described. Some additional procedures are described which can be used to alter the default structure. The binding structure establishes the path that the binder will follow when attempting to resolve external references within newly loaded user modules.

TYPEs

```
GlobalFrameHandle: TYPE = POINTER TO global FrameBase;

global FrameBase: TYPE = RECORD[
  ...,
  ownerlink: GlobalFrameHandle,
        -- points to frame of creator
  bindentry: GlobalFrameHandle,
        -- points to frame of first module to try when binding
  bindlink: GlobalFrameHandle,
        -- points to frame of next module to try when binding and current module did
        not have correct symbol and type. This is referred to as the "binding path"
        ]

NULLFrame: GlobalFrameHandle;
```

SIGNALs

InconsistentBindingPath: ERROR;
Attempt to use a **FrameHandle** that does not point to a "global" **FrameBase** was detected.

CircularBindingLink: ERROR;
Attempt to set the binding path for a global frame would have resulted in a loop in the binding path.

PROCEDUREs

Bind: PROCEDURE[f: GlobalFrameHandle];
Given the global frame **f** of a newly loaded module, this routine locates all unbound references to external objects in the module associated with **f** and attempts to bind them to objects of the same name and type that are defined in other modules. The search commences by first following the **bindentry** field of **f**. Thereafter the **bindlink** fields are

followed until all symbol references are bound or the end of the path is encountered (**f.bindlink** = NULLFrame).

BindingEntry: PROCEDURE[GlobalFrameHandle] RETURNS [GlobalFrameHandle];
Returns the current value of the **bindentry** field of the specified frame.

BindingPath: PROCEDURE[GlobalFrameHandle] RETURNS [GlobalFrameHandle];
Returns the current value of the **bindlink** field of the specified frame.

SetBindingEntry: PROCEDURE[frame, entry: GlobalFrameHandle];
sets **frame.bindentry** to **entry** after checking that all is well.

SetBindingPath: PROCEDURE[frame, entry: GlobalFrameHandle];
sets **frame.bindlink** to **entry** after checking that all is well.

Default binding paths

The MESA language construct NEW (and the debugger command "New") set up the frame of the new module so that its **bindentry** field points to itself and so that its **bindlink** field points to the frame pointed to by the **bindentry** field of its owner (the module which executes the NEW). In addition, the owner's **bindentry** field is changed to point to the new frame. Thus, if modules A, B, and C are established in order using the default mechanisms just defined, the following will be executed:

A.bindentry ← A
A.bindlink ← owner.bindentry
B.bindentry ← B
B.bindlink ← A
C.bindentry ← C
C.bindlink ← B
owner.bindentry ← C

where **owner** is the frame which corresponds to the debugger or to the module that did the NEW construct. Thus, if **Bind[C]** is called (or the debugger's Bind command is given), then the order in which binding would proceed is **B->A->owner.bindentry**. To bind B, however, the order would be **A->owner.bindentry**.

If it is desirable to have A's binding order be **C->B->A->owner.bindentry**, for example, then one would call **SetBindingEntry[A, C]** before calling **Bind[A]**.

Some additional techniques that can be used to control binding are to create an *orphan* and to *hide* a set of modules under another module. This is once again done by manipulating the **bindentry** and **bindlink** fields.

To create an orphan structure, one just sets the **bindlink** field of the first frame of the structure to NULLFrame. For example, setting **A.bindlink** to NULLFrame in the above example would make A-B-C an orphan structure. If the binding of A, B, or C is requested after the orphaning, then all of the remaining undefined references within A-B-C must be resolvable within A-B-C. In addition, no other module can be bound to any of A-B-C after it is orphaned (although they could have been bound to it prior to the orphaning).

To hide the module B under the module A, just set **A.bindentry** to B and **C.bindlink** to A. If this is done before C's undefined symbols are bound, then B will not be a candidate for binding any

of C's undefined symbols.

Alto/Mesa Debugging Documentation

December 1, 1976

<MESA-DOC>DEBUGGER

The debugging facilities being documented here are the humble beginnings of an interactive debugger for Alto/Mesa. The Mesa debugger comes in three flavors: *internal*, *external*, and *mini*. The Mesa debuggers are the primary interface between you and the Mesa runtime system. They provide facilities for loading, binding, and starting execution of Mesa programs. In addition, they provide debugging facilities which include: setting breakpoints, tracing execution, displaying the run-time state (stack), and interpreting Mesa statements. At present the interpretive facilities are severely limited. These facilities will undergo extensive development during the next few months.

Overview

The three flavors of the debugger differ in their relationship to the user program. The *internal* debugger is resident in core along with the user program being debugged. The *external* debugger resides in a different core image which is loaded when called for in very much the same way that Swat resides in a separate core image. The *mini* debugger is the code which is necessary to communicate with the *external* debugger and is resident along with the user program. The *mini* debugger also serves the function of an executive when the mesa system is first started. The *mini* debugger has only five commands each identified by their first letter. The commands will be discussed in detail below and any differences among the debuggers will be noted. For the record, the commands in the *mini* debugger are `New`, `Bind`, `Start`, `MakeImage`, `Debug`, and `Quit`. The `MakeImage` command is discussed in Section 4.

The user interface to the debugger is controlled by a command processor which invokes a collection of procedures for managing breakpoints, examining user data symbolically, and controlling the context from which user symbols are looked up. The command processor prompt character is ">" for the *mini* and *external* debuggers and "\" for the *internal* debugger (actually the character is repeated once for each nesting level of the debugger). The command syntax is tree structured and each character is extended to the maximal unique string which it specifies. Whenever an invalid character is typed, a ? is typed and you are returned to the command level. Typing a ? at any point during command selection prompts you with the collection of valid characters (in upper case) and their associated maximal strings (in lower case) and returns you to the command level. Whenever a valid command is recognized you are prompted for the parameters associated with that command (if any). Identifiers and numeric arguments are terminated with carriage-returns or spaces (the debugger will echo a delimiting character of its own choice in order to minimize loss of information from the screen). String arguments are terminated with carriage-returns. Typing rubout (delete) at any point during command selection or parameter collection returns you to the command processor.

At system start-up the *mini* debugger is given control in a context from which all the various system utilities are visible (cf. documentation for loader, binder, io-packages, etc). The only commands available are those for simple loading, binding and executing of modules. For more complex operations you must first invoke the *external* debugger with the `Debug` command. The interpretation of symbols (variable names, procedure names, etc) is based upon the following symbol-lookup algorithm. The runtime stack of procedure frames is searched in LIFO order by examining first the local frame of each procedure and then its associated global frame until a program (global) frame is encountered. The search then proceeds down the BindingPath links starting with the program frame's BindingEntry link. (For a description of binding links see the documentation for the binder.) When in doubt concerning the search order for symbol-lookup you may use the `Display Binding path` command documented below.

A. Command Tree

This is the command tree structure. Capitalized letters are typed by the user (in either upper or lower case) and the lower case substrings are echoed by the command processor. Each command is described in section B along with its parameters.

```
Display Binding path
        Module
        Frame
        Variable
        Eval-stack
        Stack
STart
 Et Context
    Octal context
    Break
    Trace
    Program Break
            Trace
Reset context
COremap [confirm]
 Reate
 Ase Ignore
     Heed
 Lear All Breaks [confirm]
          Traces [confirm]
          Entries
          Xits
      Entry Break
            Trace
      Xit Break
          Trace
      Program Break
              Trace
      Break
      Trace
Proceed [confirm]
BInd
 Reak Entry
      Xit
Trace All Entries
          Xits
      Entry
      Xit
New
Load
Octal Read
      Write
      Clear break
      Set break
Interpret Call
          @
          Pointer
          Array
          Size
          De-reference
          Expression
Quit [confirm]
↑Nstall [confirm]
```

B. Semantics of commands

The semantics of the debugger's commands are summarized below along with the parameters required by each command. The debugger prompts for all parameters. Identifiers (module, variable and type names) are accepted as a sequence of characters terminated by SPACE or RETURN. Source strings (for setting breakpoints) are accepted as a sequence of characters terminated by RETURN. A numeric parameter is a sequence of characters terminated by SPACE or RETURN which is passed to a very simple expression parser. The expression parser accepts constants in either octal or decimal and the operators +, -, *, /. Evaluation is strictly left-to-right with no precedence or parentheses allowed. All forms of numeric constants allowed by the Mesa syntax are accepted, however the default radix is octal rather than decimal. Use the "D" notation to force decimal interpretation of a numeric value. Many parameters have default values which may be used or inspected by typing ESC. After the default parameter is typed by the debugger, you may use the normal input editing conventions to modify it.

`Display Binding path`

displays the present binding path (the order which symbol-lookup searches).

`Display Module [`**module**`]`

dumps the contents of a global frame where **module** is the name of a program whose global frame has been loaded.

`Display Frame [`**address**`]`

dumps the contents of a frame where **address** is its octal address. (Useful if you have several instances of the same module.)

`Display Variable [`**id**`]`

displays the contents of a variable named **id.**

`Display Eval-stack`

displays the contents of the Mesa evaluation stack (in octal). Useful in octal debugging (ugh) or for displaying the (un-named) return values of a procedure which has been broken at its exit point.

`Display Stack`

follows down the procedure call stack (optionally) displaying the values in each frame. At each frame, the corresponding body's name, the frame's address, and the appropriate source text are displayed. You are prompted with a *">"*. A response of *V* displays all the frame's variables; *P* displays the input parameters; *R* displays the return values (those which are "named" in the RETURNS part of the body declaration); *N* displays nothing; and *Q* terminates the display and

returns you to the command processor. For a description of the output format for variables, see section C below.

STart [**address**]

starts execution of the frame whose octal address is **address.**

SEt Context [**modulename**]

changes the context (and thus the domain for symbol-lookup) to the program module whose name is **modulename.** If there is more than one instance of a module named **modulename,** the debugger lists the octal frame addresses of each instance and does not change the context. You may use the following command to set the context with a frame address.

SEt Octal context [**frame**]

changes the context (and thus the domain for symbol-lookup) to the frame whose address is `frame`. (cf. `SEt Context`).

SEt Break [**proc, text**]

sets a breakpoint in the procedure body named **proc** at the *beginning* of the statement defined by the line containing the *first* instance of the string **text.** The search for the string **text** commences at the beginning of the source text for **proc.** (Note: the range of the search extends from the beginning of the text for **proc** to the end-of-file.) When a breakpoint is encountered during execution, the debugger types the name of the body being broken, the text corresponding to that code location, and the address of the currently active frame. A nested instance of the debugger is created and control then transfers to the command processor from which you may access any of the facilities described in this document. To continue execution of your Mesa program, you execute the `Proceed` command documented below.

SEt Trace [**proc, text**]

sets a trace in the procedure body named **proc** at the *beginning* of the statement defined by the line containing the *first* instance of the string **text.** The search for the string **text** commences at the beginning of the source text for **proc.** When the tracepoint is reached, the procedure name, frame address, and source text are typed. You may respond to the "*>*" prompt with the standard replies (cf. description of `Display Stack` above) for listing the parameters, return values, or all locals. In addition to the standard replies, you may also type *B*(*b*) which will create a nested instance of the debugger and send control to the command processor.

SEt Program Break [**prog, text**]

sets a breakpoint in the program body named **prog** at the *beginning* of the statement defined by the line containing the *first* instance of the string **text.** The search for the string **text** commences at the beginning

of the source text for **prog**. (cf. SEt Break)

SEt Program Trace [**prog**, **text**]

sets a trace in the program body named **prog** at the *beginning* of the statement defined by the line containing the *first* instance of the string **text**. The search for the string **text** commences at the beginning of the source text for **prog**. (cf. SEt Trace)

Reset context

restores the context which this instance of the debugger had at its creation.

COremap [confirm]

prints the current configuration of segments in memory.

CReate [**segment**]

allocates a frame for **segment** (**segment** = SegmentHandle for code segment -- returned by Load). See the description of New below.

CAse Ignore

ignores the distinction between upper and lower case during symbol-lookup. This is the default state when you start Mesa. Upper and lower case are always different in source strings (Break and Trace).

CAse Heed

observes the distinction between upper and lower case during symbol-lookup. Once set, this state persists until you execute a CAse Ignore command.

CLear All Breaks [confirm]

clears all breakpoints.

CLear All Traces [confirm]

clears all tracepoints.

CLear All Entries [**prog**]

removes all entry traces in **prog**.

CLear All Xits [**prog**]

removes all exit traces in **prog**.

CLear Entry Break [**proc**]

converse of BReak Entry (cf. below).

Trace [**proc**]

converse of Trace Entry (cf. below).

CLear Xit Break [**proc**]

converse of BReak Xit (cf. below).

CLear Xit Trace [**proc**]

converse of Trace Xit (cf. below).

CLear Program Break [**prog, text**]

converse of SEt Program Break (cf. above).

Trace [**prog, text**]

converse of SEt Program Trace (cf. above).

CLear Break [**proc, text**]

converse of SEt Break (cf. above).

CLear Trace [**proc, text**]

converse of SEt Trace (cf. above).

Proceed [confirm]

resumes execution of the Alto/Mesa program from the point at which it was broken.

BInd [**address**]

binds the external references for the frame specified by **address**. If a symbol cannot be bound, its name is displayed on the screen. (Please refer to documentation for the binder in <MESA-DOC>BIND.BRAVO).

BReak Entry [**proc**]

inserts a breakpoint in the procedure **proc** at the first instruction after the code which stores the input parameters in **proc**'s frame.

BReak Xit [**proc**]

inserts a breakpoint at the last instruction of the procedure body for **proc**.

Trace All Entries [**prog**]

sets a trace on the entry point to each procedure in the module **prog** (cf. Trace Entry).

`Trace All Xits` **[prog]**

sets a trace on the exit point of each procedure in the module **prog** (cf. `Trace Xit`).

`Trace Entry` **[proc]**

sets a trace on the entry point to the procedure **proc.** When an entry tracepoint is encountered, **proc**'s parameters are displayed and you are prompted with "*>*". The valid responses are described in the `SEt Trace` command.

`Trace Xit` **[proc]**

sets a trace on the exit point of the procedure **proc.** When an exit tracepoint is encountered, **proc**'s return values are displayed and you are prompted with "*>*". The valid responses are described in the `SEt Trace` command.

`New` **[filename]**

equivalent to `STart[CReate[Load[`**filename**`]]]`. `Load` establishes a link between the segmentation machinery and **filename.** In addition, `Load` actually loads the object code for **filename** into memory. `CReate` allocates a frame of appropriate size for the global data space required by **filename.** The starting PC of this frame is the beginning of the code required to perform the frame's initialization. This includes both explicit user specified initialization (e.g. **x: INTEGER ← 0**) and system-initiated initialization (e.g. fabricating unique values for SIGNAL constants). `CReate` does <u>not</u> exececute this code. The first `STart` issued for this frame will execute the initialization code and return to the command processor upon completion. Subsequent `STart` commands will run the frame from the point following the initialization code and will return to the command processor upon encountering a user supplied **STOP** in the program or upon "running off the end" of the program text. At the end of each Mesa program there is an implicit **STOP** loop. Hence, attempts to `STart` a program which has "run off the end" will result in an immediate return.

`Load` **[filename]**

maps the code segment of **filename** into memory and returns a SegmentHandle (for `CReate`). See documentation for `New` above.

`Octal Read` **[address, n]**

allows you to display the **n** (octal) locations starting at **address.** (Note: these *octal* commands are being made available both as low-level debugging aids for system maintainers who must diagnose the higher-level debugging aids and system, and in lieu of expanded interpretive facilities.)

`Octal Write [address, rhs]`

allows you to store **rhs** (octal) into the location **address.** (The default for **rhs** is the current contents of **address.**)

`Octal Set break [globalframe, bytepc]`

sets a breakpoint at the byte-relative offset **bytepc** in the code segment of the frame **globalframe.**

`Octal Clear break [globalframe, bytepc]`

converse of `Octal Set break`.

`Interpret [string]`

(in the glorious future) interprets some reasonable subset of the Mesa language. For now, you may use the following interim facilities.

`Interpret Call [proc]`

calls the procedure **proc** after prompting you for the parameters one word at a time. The parameters may be octal constants, strings (enclode in double-quotes), or simple identifiers. Note: no type checking is done and string parameters are not supported by the *mini* debugger.

`Interpret @ [var]`

returns the address of **var.**

`Interpret Pointer [address, type]`

symbolically displays (according to the **type**) the value(s) stored at octal **address.** The **type** should be the type of the data rather than the type of the pointer. In searching for **type** the debugger examins first the current local frame then the corresponding global frame and its included modules.

`Interpret Array [array, index]`

displays the value of **array[index]** .

`Interpret Size [var]`

returns the number of *words* of storage allocated for **var.**

`Interpret De-reference [var]`

chase the pointer **var** one level. Note: **var** must be of type pointer.

`Interpret Expression [exp]`

evaluates exp using the simple numeric parser described and prints out the value in octal and decimal. This can be used for quick calculations or for octal-decimal conversions.

`Quit [confirm]`

returns control to the dynamically enclosing instance of the debugger (if there is one). Executing a `Quit` has the effect of cutting the runtime stack back to the nearest enclosing instance of the debugger. `Quiting` from the outermost instance of the *internal* debugger or from the *mini* debugger returns you to the Alto Executive.

`↑Nstall [confirm]`

installs the current core image as the *external* debugger. This command is invoked by typing *control-N* and is accepted only by the *internal* debugger.

C. Debugger's output conventions

The debugger uses information about the types of variables to decide on an appropriate output format. In general compile-time constants are not displayed. (Exception: `Display Variable`.) Listed below are the types which the debugger distinguishes and the convention used in each instance.

INTEGER
: always displayed in decimal. Uniformly, numeric output is decimal unless terminated by "B" (octal).

BOOLEAN
: TRUE or FALSE.

RECORD
: the record's type identifier is followed by a bracketed list of each field name and its value. E.G. v: Vector; Vector: RECORD[x,y: INTEGER]; -- the debugger displays an instance of v as `v=Vector[x: 9, y: -1]`.

ENUMERATED
: displayed as the identifier constant used in the enumerated types declaration. E.G. c: ChannelState; ChannelState: TYPE = {disconnected, busy, available}; -- the debugger displays an instance of c as c=busy. If the value is outside the range of the enumerated type (probably uninitialized variable) a ? is displayed.

STRING
: displayed as `s=(3,10)"foo"` where 3 is s's current length and 10 is its maximum

length. If the string is longer than 60 characters, only the first 40 and the last 10 are displayed.

CHARACTER
: a printing character (c) is displayed as 'c. A control character (x) other than NUL, TAB, LF, FF, CR, ESC is displayed as ↑X. Values greater than 177B are displayed in octal.

POINTER
: always displayed in octal, terminated with an "↑". E.G. p=107362B↑.

PROCEDURE, SIGNAL, ERROR
: displays the name of the procedure (signal, error) and the name of the program module in which it resides. E.G. OutChar=PROCEDURE in (DisplayIO). Procedure variables which do not contain valid procedure descriptors are flagged with a "?".

ARRAY, ARRAY DESCRIPTOR
: displays the first, second and last values of the array unless the number of elements is "small". E.G. a=(10)[Vector[x: 0, y:0], Vector[x: 1, y: 1], ... , Vector[x: 9, y:9]]. The parenthesized value to the right of the "=" is the (current) number of elements in the array.

subrange OF INTEGER
: displayed in octal if the upper limit exceeds 77777B, decimal otherwise.

D. Error messages

The debugger may generate a number of different error messages in the process of attempting to execute your commands.

`No symbol table for frame nnnnnnB`
: the symbol table file corresponding to this frame is missing and any attempt to symbolically reference variables in this module (e.g. for binding) will fail. In general, this message is only a warning and processing continues on.

`!xyz`
: the variable named xyz has not been found.

`!Number`
: an invalid number was typed.

`!File: xyz`
: the file named xyz has not been found.

`!String: xyz`
: search for the string xyz has failed.

`!String too long`
: the string you have just typed is too long. Source-text parameters are limited to 60 characters. Identifiers and string constants are limited to 40 characters.

`*** uncaught SIGNAL SoS (in MayDay)`

the user program has raised a SIGNAL (ERROR) which no one dynamically nested above the SIGNAL invocation was prepared to catch. The debugger prints the name of the SIGNAL, lists its parameters (if any), creates a new instance of the debugger, and gives control to the command processor. At this point you may, for example, trace the stack to see who raised the uncaught SIGNAL. If the semantics of the situation permit, you may proceed execution at the point of the SIGNAL's invocation by issuing a `Proceed` command. Alternatively, you retire to the dynamically enclosing instance of the debugger by issuing a `Quit` command (and lick your wounds). If the SIGNAL actually was an ERROR and you elect to `Proceed`, then prepare to read the following error message.

`ResumeError!`

you have attempted to continue execution from an ERROR. This may, of course, occur both in the situation described above or as the result of a programming error. The debugger does not support resuming SIGNALs which return values.

E. Installing the External Debugger

In order to establish the communication link between the *external* and the *mini* debuggers, you must install the *external* debugger. (This installation is similar to the installing of the Swat debugger for those familiar with this operation.) The `↑Nstall` command is invoked by typing *control-N* to the *internal* debugger which resides in the core image of the *external* debugger. The *external* debugger is contained the file XDEBUG.IMAGE (see section on getting started). To run this file you type MESA XDEBUG to the Alto Executive. This will leave you talking to the *internal* debugger (prompt character "\"). At this point you might want to load some programs to live with the *external* debugger. This is done with the commands described above. Currently the only interesting module to load is the window manager. See Section 3 for more about this module. When you are satisfied with the status of your debugger, issue the `↑Nstall` command. The command will (1) save the current core image on the file MesaDebugger and (2) exit to the Alto Executive. The debugger uses the file Swatee to hold the user core image.

Alto/Mesa Display Facilities

October 7, 1976

<MESA-DOC>DISPLAY

Introduction

The purpose of the Mesa display package is to provide the basic facilities for a wide range of user requirements in dealing with the Alto bitmap display. Since the subject of display illusions has been one of the most active and inovative, these facilities have been built in such a way as to support the currently known popular ways of dealing with the display and hopefully will provide the primitives for yet more powerful facilities. This documentation is divided into the following sections.

Bitmaps
Rectangles
Streams
Windows
Menus
Fonts

The Mesa display support modules are divided into two major sections: those that deal with the hardware or physical characteristics of the display and those that use them. Bitmaps and Rectangles are intended to provided all of the lower level facilities required to support the alto display, while Display Streams, Windows and Menus are built using those primitives. Facilities are provides for creating and manipulating bitmaps and rectangles within them, for associating streams and/or windows with rectangles, and for displaying and marking menus.

Bitmaps

The most primitive objects in the Mesa display facilities are bitmaps. A **BitmapObject** contains all the data about the physical characteristics of an actual Alto display bitmap (suitable for displaying), which is defined (in RECTANGLEDEFS.MESA) as follows:

```
BMHandle: TYPE = POINTER TO BitmapObject;

BitmapObject: TYPE = RECORD [
  link: BMHandle,          -- # NIL iff being displayed
  rectangles: Rptr,        -- list of rectangles for this map
  dcb: DCBptr,             -- address of block to be used for DCB
  addr: POINTER,           -- it's address
  words: INTEGER,          -- size of map (in words)
```

```
wordsperline: [0..maxwordsperline],
x0: xCoord,              -- x,y of upper left corner
y0: yCoord,
width: xCoord,           -- in bits(but even words)
height: yCoord,          -- in real scan lines
indenting: [0..77B],     -- in units of 16 bits
resolution: restype,
background: backgtype];
```

Initially, the system comes with a default bitmap, which is obtained by calling

```
GetDefaultBitmap: PROCEDURE
   RETURNS[BMHandle];
```

New bitmaps may be created by calling

```
CreateBitmap: PROCEDURE [pagesformap, wordsperline: INTEGER]
   RETURNS[BMHandle];
```

This procedure creates a BitmapObject, initializes it, allocates the requested space for the bitmap and initializes the map to all zeros. It also allocates a DCB for this map and initializes it in anticipation of being displayed.

A bitmap is destroyed by calling

```
DestroyBitmap: PROCEDURE [bitmap: BMHandle];
```

Before destroying a bitmap, all rectangles contained within it must be destroyed; otherwise, you will get the signal

```
BitmapError: SIGNAL [bitmap: BMHandle, error: BitmapErrorCode];
BitmapErrorCode: TYPE = {BitmapOperation};
```

Bitmap Manipulation

The following routines are provided for altering and manipulating bitmap objects in a uniform way. You may alter the size and or shape of a display bitmap by calling

```
ReallocateBitmap: PROCEDURE
      [bitmap: BMHandle, pagesformap, wordsperline: INTEGER];
```

This procedure may also be used to deallocate the memory used for a bitmap and subsequently reallocate it.

If you decide to alter any of the fields of a bitmap object and wish to have the effect of that alteration reflected in the hardware (e.g. change background from white to black) call

```
UpdateBitmap: PROCEDURE [bitmap: BMHandle]
   RETURNS[DCBptr];
```

The above call returns the real DCB address for the specified **BitmapObject** because **DCB's** (as required by the Alto harware) must be even word alligned and the field in the **BitmapObject** may or may not be alligned.

The following two procedures are supplied to actually display (undisplay) a bitmap.

```
DisplayBitmap: PROCEDURE [bitmap: BMHandle];

UnDisplayBitmap: PROCEDURE [bitmap: BMHandle];
```

Rectangles

Rectangles describe arbitrary rectangular regions within a bitmap. Together with bitmaps, rectangles and the procedures for manipulating them implement the most primitive functions of the display package. (Definitions are in RECTANGLEDEFS.MESA)

```
Rptr: TYPE = POINTER TO Rectangle;

Rectangle: TYPE = RECORD
    [
    link: Rptr,
    visible: BOOLEAN,
    options: RectangleOptions,
    bitmap: BMHandle,
    x0, width, cw: xCoord,      -- relative to bitmap origin
    y0, height, ch: yCoord
    ];

RectangleOptions: TYPE = RECORD
    [
    NoteInvisible: BOOLEAN,     -- SIGNAL if rectangle off bitmap
    NoteOverflow: BOOLEAN,      -- SIGNAL if attempt to store outside
    ];
```

The system is intialized with a default rectangle that encompasses the entire default bitmap; its identity is returned by

```
GetDefaultRectangle: PROCEDURE
   RETURNS[Rptr];
```

New rectangles are created by calling the procedure

```
CreateRectangle: PROCEDURE
[bitmap: BMHandle, x0, width: xCoord, y0, height: yCoord]
   RETURNS[Rptr];
```

Writing of text at arbitrary locations within a rectangle is accomplished by

```
WriteRectangleChar: PROCEDURE
  [rectangle: Rptr, x: xCoord, y: yCoord, char: CHARACTER, pfont: FAptr]
   RETURNS[xCoord, yCoord];

WriteRectangleString: PROCEDURE
  [rectangle: Rptr, x: xCoord, y: yCoord, str: STRING, pfont: FAptr]
   RETURNS[xCoord, yCoord];
```

Where x and y are relative to the *rectangle* origin. If the rectangle is not visible (e.g. outside the bounds of the bitmap) or x, y is outside the rectangle then one of the SIGNALs **NotVisible**, **RightOverflow** or **BottomOverflow** is generated.

The following procedures will handle clipping situations for rectangle overflow either to the right or off the bottom. In the current implementation they will not allow a rectangle to either go off the top or to the left of a bitmap.

Rectangles may be moved or their size altered by invoking

```
MoveRectangle: PROCEDURE [rectangle: Rptr, x: xCoord, y: yCoord];

GrowRectangle: PROCEDURE [rectangle: Rptr, width: xCoord, height: yCoord];
```

Where **x**, **y**, **width** and **height** are relative to the *bitmap* origin.

Rectangle Utilities

The following procedures implement commonly used operations on rectangles. They are by no means a complete set but are simply the ones used in providing the basic Mesa system facilities.

Coordinates are relative to the *rectangle* origin.

```
DrawBoxInRectangle: PROCEDURE
    [rectangle: Rptr, x0, width: xCoord, y0, height: yCoord];
```

Draws a rectangular box with lines of width one inside the supplied rectangle.

```
ScrollBoxInRectangle: PROCEDURE
    [rectangle: Rptr, x0, width: xCoord, y0, height: yCoord, incr: INTEGER];
```

Will scroll the rectangular region within the supplied rectangle defined by **x0**, **y0**, **width**, and **height** either up or down as specified by **increment** (+ = up).

```
InvertBoxInRectangle: PROCEDURE
    [rectangle: Rptr, x0, width: xCoord, y0, height: yCoord];
```

Will *video reverse* the rectangular region within the supplied rectangle defined by **x0**, **y0**, **width**, and **height**.

ClearBoxInRectangle: PROCEDURE
[rectangle: Rptr, x0, width: xCoord, y0, height: yCoord, gray: GrayPtr];

Will *clear* the rectangular region within the supplied rectangle defined by **x0, y0, width,** and **height** to the supplied *gray* pattern (e.g. 0 = clear -1 = black etc.).

The display facilities allow you to alter both the bitmap and the position of a rectangle within a bitmap such that it is possible for a rectangle to be *not visible* (entirely outside the bounds of the bitmap). You may determine if a rectangle is visible by calling

IsRectangleVisible: PROCEDURE [rectangle: Rptr]
RETURNS[BOOLEAN];

Exceptional Conditions

The following exception conditions are optionally reported to the user based upon the setting of the the rectangle object flags **NoteInvisible** and **NoteOverflow.**

RectangleError: SIGNAL [rectangle:Rptr, error: RectangleErrorCode];
RectangleErrorCode: TYPE = {RightOverflow, BottomOverflow, NotVisible};

Coordinate Conversion Routines

The following procedures allow you to convert between cursor, rectangle and bitmap coordinates. They worry about indenting and other displayed bitmaps.

CursorToMapCoords: PROCEDURE [bitmap: BMHandle, x: xCoord, y: yCoord]
RETURNS[xCoord, yCoord];

Converts the cursor coordinates (display origin relative) to bitmap coordinates (bitmap origin relative).

RectangleToMapCoords: PROCEDURE [rectangle: Rptr, x: xCoord, y: yCoord]
RETURNS[xCoord, yCoord];

Converts rectangle coordinates (rectangle origin relative) to bitmap coordinates (bitmap origin relative).

Display Streams

Display streams are provided in Mesa to perform teletype simulation operations that are commonly associated with display based systems. Display streams are associated with a previously created rectangle at stream creation time. For a more complete desciption of streams see the STREAMS documentation. The system comes equipped with a default displaystream. Interpretation of the standard stream operations is as follows:

reset[s] clears the rectangle associated with s.

get[s] produces a **StreamAccess** error.

putback[s,i] produces a **StreamAccess** error.

put[s,i] *writes* the CHARACTER i in the next character position. Options are provided for line wrapping/truncation and scrolling/discarding.

endof[s] produces a **StreamAccess** error.

destroy[s] destroys s in an orderly way, freeing the space it occupies. If s is the default display stream, the **StreamOperation** error results.

Initially, the default display stream is defined by a rectangle which occupies the entire default system bitmap, the stream's handle is returned by the procedure

```
GetDefaultDisplayStream: PROCEDURE RETURNS [StreamHandle];
```

Any number of display streams may be created, using the procedure

```
CreateDisplayStream: PROCEDURE [rectangle: Rptr]
  RETURNS [DisplayHandle];
```

where **rectangle** is a pointer to a rectangle object associated with a bitmap. It should be noted that display streams by themselves provide no facilities for dealing with retangles that *overlap* (e.g. the characters are simply OR'ed together) however, facilities are provided by *windows* for dealing with this situation in an orderly manner.

The following procedures implement *backspace* character and line functions.

```
ClearDisplayChar: PROCEDURE [stream: StreamHandle, char: CHARACTER];

ClearCurrentLine: PROCEDURE [stream: StreamHandle];

ClearDisplayLine: PROCEDURE [stream: StreamHandle, line: INTEGER];
```

The requirement to pass the character to be erased in ClearDisplayChar is a little hoaky, but this is necessary because the stream retains no memory of what characters were displayed.

Windows

Windows provide a uniform mechanism for managing the data (text or bit arrays) contained in rectangles. Windows and the system supplied procedures are designed to allow the user to recreate or reposition a view in a rectangle in a standard manner.

(Definitions are in WINDOWDEFS.MESA)

```
WindowType: TYPE = {clear, random, scratch, file, scriptfile};

WindowHandle: TYPE = POINTER TO DisplayWindow;

DisplayWindow: TYPE = RECORD
    [
    link: WindowHandle,
    type:  WindowType,
    name: STRING,
    menu: MenuHandle,
    displayproc: PROCEDURE [WindowHandle],
    rectangle: Rptr,
    ds: DisplayHandle,
    ks: KeyHandle,
    file: DiskHandle,
    fileindex: StreamIndex,
    tempindex: StreamIndex,
    eofindex: StreamIndex,
    selection: Selection
    ];

  Selection: TYPE = RECORD
    [
    ptr1: StreamIndex,
    ptr2: StreamIndex
    ];
```

New display windows may be created by calling

```
CreateDisplayWindow: PROCEDURE
   [type: WindowType, r: Rptr, ds: DisplayHandle, ks: KeyHandle, name: STRING]
     RETURNS[WindowHandle];
```

Conversly, display windows may be destroyed and all data associated with them released by invoking

```
DestroyDisplayWindow: PROCEDURE [w: WindowHandle];
```

It is often desirable to be able to transform one type of window into another (e.g. load a file into a scratch window). The following procedure undoes the old attributes and sets up new ones.

```
AlterWindowType: PROCEDURE
   [w: WindowHandle, type: WindowType, name: STRING];
```

Refreshing both the border and content of a window is accomplished via the following procedure. Actual content refreshing is accomplished by dispatching to a user supplied

(or system default) procedure.

PaintDisplayWindow: PROCEDURE [w: WindowHandle];

The concept of a window being *current* (e.g. on top) is central to this implimentation of display windows. Windows are maintained in a ring in the order they were last *current*. To make a window *current* (which also refreshes its content) call

SetCurrentDisplayWindow: PROCEDURE [w: WindowHandle];

At any point you may determine which window is *current* by calling

GetCurrentDisplayWindow: PROCEDURE
RETURNS[WindowHandle];

To determine which window (if any) the cursor may be in call

FindDisplayWindow: PROCEDURE [x: xCoord, y: yCoord]
RETURNS[WindowHandle, xCoord, yCoord];

Coordinates are cursor coordinates (display origin relative). If **WindowHandle** is **NIL** then the supplied x, y are not in any window. If a non-NIL WindowHandle is returned then the x, y are converted into rectangle coordinates (rectangle origin relative) and returned also.

For file type windows you may alter the file being displayed by invoking

SetFileForWindow: PROCEDURE [w: WindowHandle, name: STRING];

Scratch, scriptfile and file type windows use a Stream to manage the data contents of the window. You may reposition the displayed contents of these windows by calling

SetIndexForWindow: PROCEDURE [w: WindowHandle, index: StreamIndex];

SetPositionForWindow: PROCEDURE [w: WindowHandle, position: INTEGER];

Exceptional Conditions

In general the procedures implementing windows do not generate SIGNALs or ERRORs but rather attempt to muddle through whatever nonsense you supplied or tried to do. This results in coercing coordinates and turning funny calls into NOPs in most cases.

Menus

Menus supply a uniform command specification facility. The current implementation is very simple and just provides for a correspondence between a keyword and a procedure to be invoked if that key word is selected. The display algorithm and selection technique are built-in. (Definitions are in MENUDEFS.MESA)

MenuHandle: TYPE = POINTER TO MenuObject;

```
MenuObject: TYPE = RECORD
  [
  link: MenuHandle,
  index: INTEGER,
  width: xCoord,
  rectangle: Rptr,
  menuseg: SegmentHandle,
  dataseg: SegmentHandle,
  array: MenuArray
   ];

MenuArray: TYPE = DESCRIPTOR FOR ARRAY OF MenuItem;

MenuItem: TYPE = RECORD
    [
    keyword: STRING,
    proc: MenuProc
    ];
```

To satisfy the Mesa TYPE checker, all menu procedures must have a uniform calling sequence, which is as follows

```
MenuProc: TYPE = PROCEDURE [w: WindowHandle, x: xCoord, y: yCoord];
```

New MenuObjects are created and destroyed by calling

```
CreateMenu: PROCEDURE [array: MenuArray, width: xCoord]
  RETURNS[MenuHandle];

DestroyMenu: PROCEDURE [menu: MenuHandle];
```

A menu is actually displayed on the Alto display screen by calling

```
DisplayMenu: PROCEDURE
  [menu: MenuHandle, bitmap: BMHandle, x: xCoord, y: yCoord];
```

Coordinates are bitmap origin relative. This procedure saves the contents of the bitmap to be overlayed so it may be restored later.

To take a menu down and restore the previous contents of the bitmap call

```
ClearMenu: PROCEDURE [menu: MenuHandle];
```

Items in a displayed menu are marked as selected by invoking

```
MarkMenuItem: PROCEDURE [menu: MenuHandle, index: INTEGER];
```

The current implementation assumes that only one menu item will be marked at a time. Therefore simply marking a new item will *unmark* the currently marked item (if any). Marking is currently done by video reversal.

Exceptional Conditions

There is currently no checking to ensure that you do not attempt to clear non-displayed menus or other such nonsense. If you do, you are on your own.

Fonts

Font procedures assume the standard Alto font format. Only the most simple font procedures are supplied as follows.

```
ComputeCharWidth: PROCEDURE [char: CHARACTER, font: FontHandle]
    RETURNS [INTEGER];
```

```
GetDefaultFont: PROCEDURE
    RETURNS [font: FontHandle, lineheight: INTEGER];
```

Returns the system font.

```
GetFont: PROCEDURE [SegmentHandle];
    RETURNS [font: FontHandle];
```

Swapin and locks the font segment.

```
GetFontSegment: PROCEDURE [filename: STRING]
    RETURNS [SegmentHandle];
```

Allocates space and creates a Segment for a font.

Alto/Mesa File Machinery

December 1, 1976

<MESA-DOC>FILES

Introduction

Logically, the Mesa file package is a sub-module of the segmentation machinery, but it is described separately because other objects (e.g. disk streams) also use this interface. Internally, the file machinery maintains a set of items called **FileObjects**: a file object, among other things, contains the file's disk address and serial number, as well as its access rights, several reference counts, and a file length hint.

The Mesa system follows most conventions of the Alto file system (although some, including multiple versions, sub-directories, and the system log are not currently supported). See the Bcpl Operating System document for a description of the Alto file system. A description of the various procedures used to manipulate the Alto's directory appears at the end of this section.

Files

A file is an integral number of pages which logically appear to be contiguous, irrespective of their physical location. The pages of a file are numbered from zero up to some maximum:

```
MaxFilePage: CARDINAL;  -- maximum file page number

PageNumber: TYPE = [0..MaxFilePage];
```

In the Alto file system, page zero of the file (the leader page) is special; it contains file status information. Thus the data actually begins at page one.

Externally, a file is known by its name, which is just a string. Internally Mesa retains only a file's FP, which is an abreviated form of the Alto file system's file pointer:

```
FP: TYPE = RECORD [
  serial: SN,  -- internal file serial number
  leaderDA: vDA];  -- first virtual disk address
```

The correspondence between file names and FPs is maintained in the file system's directory (SysDir). After the file is initially looked up in this directory, the name is discarded; the Mesa world deals only in FPs thereafter. A directory search is required if the name must be recovered.

File Objects

A **FileHandle** is used to refer to a file in the Mesa environment, and can be obtained by a call on NewFile; it is simply a pointer to a record called a **FileObject.**

```
FileHandle: TYPE = POINTER TO FileObject;

FileObject: TYPE = RECORD [
  open: BOOLEAN,          -- if the file is open
  read, write, append: BOOLEAN,    -- access rights
  lock: [0..MaxRefs],  -- reference count
  segcount: [0..MaxRefs],  -- attached segments
  swapcount: [0..MaxRefs],  -- swapped in segments
  . . . ];              -- plus other private fields
```

The following options are used when creating new file objects (these will become set types):

```
AccessOptions: TYPE = [0..7];
  Read: FileOptions = 1;
  Write: FileOptions = 2;
  Append: FileOptions = 4;

VersionOptions: TYPE = [0..3];
  NewFileOnly: FileOptions = 1;
  OldFileOnly: FileOptions = 2;
```

Read access allows existing pages of the file to be read; **Write** means that existing pages can be written (or deleted; perhaps a separate **Delete** option should be included). **Append** allows new pages to be added to the end of the file (files do not have holes in them). Note that **Append** does *not* imply **Write** access.

Disallowed combinations are {**NewFileOnly**, **OldFileOnly**} and {**NewFileOnly**, ~**Append**}. If **Append** access is not specified, **OldFileOnly** is assumed. If you like, you may specify **DefaultAccess** or **DefaultVersion**, which are equivalent to **Read** and create the file if necessary.

Signals associated with file objects:

```
FileNameError: SIGNAL [name: STRING];
```

> The file name is invalid, or the file does not exist (**OldFileOnly**), or the file does exist (**NewFileOnly**).

```
FileAccessError: SIGNAL [file: FileHandle];
```

> An attempt to perform some operation not allowed by the current access, or the requested access and version options are inconsistent (see the diallowed combinations above).

```
InvalidFP: SIGNAL [fp: POINTER TO FP];
```

> A file positioning operation has determined that the file serial number in the **FP** of the file object does not match the disk label.

FileError: SIGNAL [file: FileHandle]; all other file errors

A **FileObject** is created using the following procedures:

NewFile: PROCEDURE [
name: STRING, access: AccessOptions, version: VersionOptions]
RETURNS [FileHandle];

> Given a file name and access rights, this procedure creates a new file object and returns a pointer to it. A check is made that the file exists in the directory, creating it if necessary, but the file is not opened as a result of this call. Objects attached to the file (segments and streams, for example) ensure that the file is open before attempting a transfer. If there is already a file object for the file specified, its access is updated (by or'ing; this is not a protection system), and a pointer to the old object is returned.

InsertFile: PROCEDURE [
fp: POINTER TO FP, access: AccessOptions]
RETURNS [FileHandle];

> Creates a file object directly from **fp**, without searching any directory. If there is already a file object with a matching **fp**, its access is updated (by or'ing; this is not a protection system), and a pointer to the old object is returned.

Internally, Mesa keeps track of the number of segments attached to each file (**segcount**) and of those the number which are currently swapped in (**swapcount**). When the swap count goes to zero, the file may be closed, and when the segment count goes to zero, the file object is released (only the latter operation happens automatically). Since a file may have other objects attached to it (streams, for example), it may be necessary to prevent the file object from being released even when there are no more segments attached to it. The **lock** field serves this purpose, and is manipulated by the procedures

LockFile, UnlockFile: PROCEDURE [file: FileHandle];

A maximum of **MaxRefs** locks may be performed on each file object. Note that a file object is *not* automatically released when its lock count goes to zero.

A **FileObject** is released by

ReleaseFile: PROCEDURE [file: FileHandle];

> The file is first closed if it is open; then its file object is released. A **FileError** will be generated if there are segments associated with the file at the time of this call. *Note:* except for this error check, releasing a file which is locked is a no-op.

A file is physically destroyed by calling

DestroyFile: PROCEDURE [file: FileHandle];

> In addition to releasing the file object, the file's pages are deleted and its entry is removed from the directory. The file object must not have any segments currently

attached to it, nor may it be locked; either condition results in a **FileError.**

To be on the safe side, destroying a file is somewhat complicated if it currently has segments attached to it. The file must first be locked, then all of its segments deleted, then the file should be unlocked and finally **DestroyFile** should be called. This sequence assumes that no other user has a **Lock** on the file.

Characteristics of the disk file associated with a **FileObject** are obtained and changed using the following procedures:

FindFile: PROCEDURE [fp: POINTER TO FP] RETURNS [FileHandle];

Searches all existing file objects for one whose serial number and disk address match those contained in **fp.** Returns NIL if no match can be found.

GetFileFP: PROCEDURE [file: FileHandle, fp: POINTER TO FP];

Copies the file pointer from **file** into **fp.**

GetFileAccess: PROCEDURE [file: FileHandle] RETURNS [access: AccessOptions];

Converts the **read, write,** and **append** bits of a file object into a form that can be passed to **NewFile.**

SetFileAccess: PROCEDURE [file: FileHandle, access: AccessOptions];

Or's access into the file object (this is not a protection system).

GetEndOfFile: PROCEDURE [file: FileHandle]
RETURNS [page: PageNumber, byte: CARDINAL];

Returns the page number of the last page in the file that contains data, together with the number of bytes in that page (the number of the first non-existant byte in the page, counting from zero). In the Alto file system, page zero is the leader page, the first data page being page one. Note that if the last data page is full, a null page is appended to the file, but **GetEndOfFile** does *not* tell you about it (so do not count on it being there). For an empty file, this routine returns **[0, BytesPerPage]** (reflecting the existence of the leader page).

GetEndOfFile first opens the file (if it is closed) to obtain the length hint from the leader page. It also inserts the current file length into the file object, so that subsequent requests for the length will not require reading the disk.

SetEndOfFile: PROCEDURE [file: FileHandle, page: PageNumber, byte: CARDINAL];

Extends or truncates the file as necessary to make **page** the number of its last data page, with **byte** bytes in it. The arguments are first adjusted to include a null page if necessary. Extending requires **Append** access, truncating requires **Write** access.

The procedure **EnumerateFiles** is provided so that one may conveniently scan all file objects that currently exist.

EnumerateFiles: PROCEDURE [

proc: PROCEDURE [FileHandle] RETURNS [BOOLEAN]]
RETURNS [FileHandle];

This procedure calls **proc** once for each file object that is currently in use. This process will terminate when the list of file objects is exhausted or when **proc** returns **TRUE**. In the latter case, the **FileHandle** of the last **FileObject** processed is returned. Otherwise, **NIL** is returned.

If new file objects are created while **EnumerateFiles** is in control, it is not guaranteed that they will be included in the sequence of **FileHandles** passed to **proc**.

Directories

A number of procedures are provided to manipulate the Alto directory (SysDir). Currently, these routines do *not* support either sub-directories or file version numbers. The simplest operation provided is to enumerate all of the file entries in the directory:

EnumerateDirectory: PROCEDURE [
proc: PROCEDURE [POINTER TO FP, STRING] RETURNS [BOOLEAN]];

This procedure calls **proc** once for each filename in the directory, passing it pointers to the file's FP and name. These parameters are local to the enumeration procedure and must be copied if they are to be retained after the enumeration completes. Processing terminates when **proc** returns **TRUE** or the end of the directory is reached.

The following procedures may be of use to programmers implementing features beyond those provided by **NewFile** and **DestroyFile**.

DirectoryLookup: PROCEDURE [
fp: POINTER TO FP, name: STRING, create: BOOLEAN] RETURNS [old: BOOLEAN]

This procedure looks up **name** in the directory. If an entry already exits, its file pointer is copied into **fp** and **TRUE** is returned, otherwise **FALSE** is returned. In addition, if **create** is **TRUE**, the file will be created (with one empty data page), and the new file pointer will be copied into **fp**.

DirectoryLookupFP: PROCEDURE [
fp: POINTER TO FP, name: STRING] RETURNS [old: BOOLEAN]

This routine is similar to **DirectoryLookup**, except that the directory is searched for a matching FP. It returns **TRUE** if the file pointer was found. In addition it will supply the filename if **name** is not **NIL**.

DirectoryPurge: PROCEDURE [
fp: POINTER TO FP, name: STRING] RETURNS [found: BOOLEAN]

This procedure removes the entry corresponding to **name** from the directory (it does *not* disturb the file pages pointed to by the FP, however). If the file is found, its file pointer is copied into **fp** and **TRUE** is returned, otherwise **FALSE** is returned.

DirectoryPurgeFP: PROCEDURE [

fp: POINTER TO FP] RETURNS [found: BOOLEAN]

This routine is similar to **DirectoryPurge**, except that the directory is searched for a matching **FP**. It returns **TRUE** if the file pointer was found, deleting the entry in the process. Perhaps it should also copy the file's name into a supplied parameter?

Alto/Mesa Process Package

May 19, 1976

<MESA-DOC>PROCESS

<MESA>PROCESS.MESA contains procedures that implement a simple process switching facility. The necessary declarations appear in <MESA>PROCESSDEFS.MESA and are described below.

MACHINE CODEs

Logically this instruction is a procedure, but may not be assigned to procedure variables.

BLOCK: MACHINE CODE

This instruction makes the current process not ready and runs the highest priority ready process. A process should only block when there is some external condition which will make it ready again. The external condition might be another process on an asynchronous interrupt.

TYPEs

ProcessObject: TYPE = PRIVATE RECORD [. . .];
ProcessHandle: TYPE = POINTER TO ProcessObject;

A ProcessObject describes the state of a suspended process. The state consists of the evaluation stack and stack pointer, a pointer to the frame, and the priority of the process. The constant ProcessNIL points to the null process.

ProcessPriority: TYPE = [0..15]
HighestProcessPriority: ProcessPriority=2
LowestProcessPriority: ProcessPriority=14

The priority of a process determines the order in which ready processes are run. Because of limitations of the Alto implementation of process switching, the range of useable priorities is limited to [2..14]. In the standard system, the keyboard handler runs at priority 2 and the Debugger at priority 14. The priority of a process may be **Unscheduled** meaning that the process is not in the process switching system, but has not been destroyed.

SIGNALs

PriorityNotAvailable: ERROR;

A requested priority slot is in use.

InvalidPriority: ERROR;

A priority is not in the proper range.

InvalidProcess: ERROR;

An object does not appear to be a proper process.

ProcessNotScheduled: ERROR;

Attempt to change active or ready for a process whose priority is Unscheduled.

PROCEDUREs

DisableInterrupts, EnableInterrupts: PROCEDURE

These procedures disable and enable the process switching mechanism. They should surround any (hopefully small) segments of code manipulating writeable data which may be accessed by more than one active process.

CreateProcessFromFrame: PROCEDURE [FrameHandle, ProcessPriority]
RETURNS [ProcessHandle]

CreateProcessFromProcedure: PROCEDURE [PROCEDURE, ProcessPriority]
RETURNS [ProcessHandle]

These procedures create ProcessObjects allocating the necessary space from the heap. The process may be a procedure or a frame. If the process ever "returns" it winds up executing BLOCK endlessly.

SetProcessPriority: PROCEDURE [ProcessHandle, ProcessPriority]

GetProcessPriority: PROCEDURE [ProcessHandle]
RETURNS [ProcessPriority]

These procedures set or read the priority field of a process object. When the priority of a process is set, it is made unready and inactive. The priority may be set to Unscheduled.

GetCurrentProcess: PROCEDURE RETURNS [ProcessHandle]

GetCurrentPriority: PROCEDURE RETURNS [ProcessPriority]

These procedures return information about the currently running process.

DestroyProcess, MakeProcessReady, ActivateProcess,
DeActivateProcess: PROCEDURE [ProcessHandle]

These procedures operate on processes. DestroyProcess returns the space in the process object to the heap. The other procedures simply control the process switching mechanism. In order to run, a process must be both ready and active. A process is also made ready by the occurrance of a Nova interrupt on the channel corresponding to its priority.

EnumerateProcess: PROCEDURE [PROCEDURE [ProcessHandle] RETURNS [BOOLEAN]]
RETURNS [ProcessHandle]

The passed procedure is invoked with processes in order of decreasing priority. If the value of the procedure is TRUE EnumerateProcess returns the last ProcessHandle. Otherwise ProcessNIL is returned.

Alto/Mesa Segment Machinery

December 1, 1976

<MESA-DOC>SEGMENTS

Introduction

The Mesa virtual memory (VM) is organized as a vector of pages of size **PageSize** words; the last page is **MaxVMPage**. VM is occupied by segments: a segment is an integral number of pages in length and the words in a segment are all linearly addressable -- i.e., segments have no empty holes in them. *Data* segments are associated directly with memory and are not swappable or movable; *file* segments correspond to contiguous groups of pages in a file and may be swapped in and out of virtual memory.

User programs access file segments using **FileSegmentHandles**, which are pointers to **FileSegmentObjects**. A **FileSegmentObject** contains sufficient information to compute its address if the segment is swapped in. Internally, the segmentation package maintains a set of objects called **FileObjects**: a file object, among other things, contains the file's disk address and serial number, as well as its access rights. The association between a segment and a file is made when the segment is created. The Mesa file package is documented separately.

Segments may also be pages in VM rather than being attached to a file. Such data segments are not swappable or movable in any way (relative to the Mesa virtual memory). Thus, absolute pointers into a data segment are valid for the lifetime of the segment. **DataSegmentHandles** and **DataSegmentObjects** are used to record information about these segments.

Data Segments

As mentioned above, segment objects come in two varieties: data segments and file segments. Data segments are associated only with virtual memory (there is no swapping file), and are never moved or swapped out.

```
DataSegmentHandle: TYPE = POINTER TO DataSegmentObject;

DataSegmentObject: TYPE = RECORD [
  . . . ,     -- other private fields
  pages: [1..MaxVMPage+1],  -- number of pages
  VMpage: [0..MaxVMPage]];  -- location in VM
```

The procedures which manipulate data segments are:

```
NewDataSegment: PROCEDURE [
  base: PageNumber, pages: PageCount]
  RETURNS [DataSegmentHandle];
```

Create a new data segment and return a handle for it. If **base** is **DefaultBase** then the segment is allowed to begin on any free page in memory. If **base** is an actual page number (in [0..MaxVMPage]), an attempt is made to place the segment at that location. Note that **pages** should not be defaulted.

VMNotFree: SIGNAL [base: PageNumber, pages: PageCount];

In **NewDataSegment**, the base was not **DefaultBase** and the specified memory pages were not free.

DataSegmentAddress: PROCEDURE [seg: DataSegmentHandle] RETURNS [POINTER];

Returns a pointer to the base of the segment in virtual memory. In the current implementation, segments always begin on a page boundary; this may not be true in the (distant) future.

VMtoDataSegment: PROCEDURE [a: POINTER] RETURNS [DataSegmentHandle];

The handle for the segment containing the specified address (as currently laid out in memory) is returned. NIL is returned if no data segment contains it. This does not imply that the page containing the address is free, however; it may be assigned to a file segment, or reserved for some operation currently in progress. Note: this operation requires a search of all existing data segment objects.

DeleteDataSegment: PROCEDURE [seg: DataSegmentHandle];

The specified data segment is deleted and its segment object freed. When a segment is successfully deleted, any VM which it occupied becomes free.

```
EnumerateDataSegments: PROCEDURE [
  proc: PROCEDURE [DataSegmentHandle] RETURNS [BOOLEAN]]
  RETURNS [DataSegmentHandle];
```

proc is called once for each data segment currently defined in the system. If **proc** returns TRUE, **EnumerateDataSegments** returns the **DataSegmentHandle** of the last segment processed. If the end of the set of data segments is reached, NIL is returned.

If new data segments are created while **EnumerateDataSegments** is in control, it is not guaranteed that they will be included in the sequence of **DataSegmentHandles** passed to **proc**.

File Segments

Unlike data segments, file segments are associated with a contiguous group of pages in a file and are therefore swappable. Pointers into a file segment are valid only while it is swapped in (and locked so that it will not be swapped out). A file segment which is swapped out occupies no space in virtual memory other than the segment object which describes it.

FileSegmentHandle: TYPE = POINTER TO FileSegmentObject;

```
FileSegmentObject: TYPE = RECORD [
  swappedin: BOOLEAN,  -- TRUE iff segment is swapped in
  read, write: BOOLEAN,  -- access options
  class: FileSegmentClass,  -- user settable (see below)
  lock: [0..MaxRefs],  -- locking reference count
  file: FileHandle,  -- see the file machinery
  base: PageNumber,  -- first page of the file to include
  pages: [1..MaxVMPage+1],  -- number of pages, beginning with base
  VMpage: [0..MaxVMPage],  -- if swapped in, VM page number
  . . . ];  -- plus other private fields

FileSegmentClass: TYPE = {code, symbols, other};
```

To create new file segments, use

```
NewFileSegment: PROCEDURE [
  file: FileHandle, base: PageNumber, pages: PageCount, access: AccessOptions]
  RETURNS [FileSegmentHandle];
```

> Creates a new segment and returns a handle for it. The segment is associated with the corresponding file pages, but the file is not opened and the segment is not swapped in. If base is DefaultBase, the segment will begin with the first data page of the file, and if pages is DefaultPages, it will include the last page of the file. Although it is generally not done, a segment can begin with the leader page (page zero) of a file. Finally, if access is DefaultAccess, read access is assumed.

If the access specifies that changing the data is permitted, then whenever it is necessary to swap this segment out and remove its pages from memory, pages will be written back to the file (the Alto has no hardware to detect if the pages have actually been changed). Note that it is possible to change the segment's access (by setting the write bit, for example), provided the file to which it is attached has the appropriate access rights.

```
InvalidSegmentSize: SIGNAL [pages: PageCount];
```

> This signal is generated whenever a zero length segment is requested, or if the length exceeds the size of virtual memory.

```
FileSegmentAddress: PROCEDURE [seg: FileSegmentHandle] RETURNS [POINTER];
```

> The address of the beginning of the segment is returned. NIL is returned if the segment is not currently swapped in. To guarantee the validity of the address, the segment should be locked when this procedure is called (see below), since the system may swap out file segments which are not locked. *Beware of dangling references!*

```
VMtoFileSegment: PROCEDURE [a: POINTER] RETURNS [FileSegmentHandle];
```

> The handle of the file segment containing the specified address (as currently laid out in memory) is returned. NIL is returned if no file segment contains it (this does not imply that the page containing the address is free, however; it may be assigned to a data segment, or reserved for a segment whose swap in is in progress). Note: this operation requires a search of all existing file segment objects.

```
DeleteFileSegment: PROCEDURE [seg: FileSegmentHandle];
```

The specified file segment is deleted and its segment object is released. If the segment is swapped in, it is first swapped out (it should not be locked). If there are no other segments associated with this segment's file (the file's **segcount** is zero), then **ReleaseFile** is called to release the **FileObject**. When a segment is successfully deleted, any VM which it may have occupied becomes free.

Window Segments

A window segment is similar to a file segment except that the **base** and **pages** fields of the segment may be altered after it is created, in order to slide the window around in a file or to vary the window's size. In reality, all file segments are in fact window segments, and may be moved with the following procedure:

MoveFileSegment: PROCEDURE [
seg: FileSegmentHandle, base: PageNumber, pages: PageCount];

If the segment is swapped in, it is first swapped out (it should not be locked). The segment is then moved to the new location in the segment's file, but it is not swapped in. The **base** and **pages** are defaulted as in **NewFileSegment**. In the current implementation, the disk address of the original segment position is retained as a hint about the the new location, thus improving performance considerably when a one page segment is slid forward or backward in a file.

If the original and final position of the segment overlap, there is no guarantee that the overlapping pages are actually written, nor is it guaranteed that a minimum number of pages are transferred. The segment machinery reserves the right to implement (or to unimplement) such optimizations in the future.

Swapping Segments

A segment can be swapped into and out of VM. The procedures and signals which implement this are

SwapIn: PROCEDURE [seg: FileSegmentHandle];

Swap in the specified segment (if it is swapped out), opening the associated file if necessary. Lock it so it won't be moved or swapped out. A **SwapError** will result if the segment already has **MaxRefs** locks on it, or if the segment's file has **MaxRefs** segments currently attached to it and swapped in.

InsufficientVM: SIGNAL [pages: PageCount];

There is not enough contiguous memory to accomodate a segment; **pages** is the number of pages that are actually required. If resumed, the allocation will be retryed; this gives the catcher of this signal a chance to free up some VM. Users can free VM pages by deleting data segments and by allowing locked segments to become swappable (see also the section below on swapping strategies).

SegmentFault: SIGNAL [seg: FileSegmentHandle, pages: PageCount];

> End of file was encountered while attempting to swap the segment in or out; **pages** is the actual number of pages in the segment. If **pages** is greater than zero then the signal may be resumed and the segment will be truncated (of course, this will not alter the file length).

To unlock a segment (allow it to be swapped), use the procedure:

Unlock: PROCEDURE [seg: FileSegmentHandle];

> Unlock the specifed segment so that it can be swapped out. Note that locking behaves like reference counting, so that locks (performed by **SwapIn**) must be properly paired with **Unlocks**.

A segment is swapped out using

SwapOut: PROCEDURE [seg: FileSegmentHandle];

> Swap out the specified segment, writing the pages back to the file if the segment's access makes this necessary, and free the segment's VM pages. If the segment is locked, a **SwapError** will be generated.

A program may explicitly request that the file pages corresponding to a segment be updated by calling

SwapUp: PROCEDURE [seg: FileSegmentHandle];

> Write the pages of the segment back to the file if the access requires it, but do not unlock the segment or free the segment's VM pages.

Note that neither **SwapIn**, **SwapOut**, or **SwapUp** are capable of extending a file (physically adding pages or bytes to it) based on the size of a segment. Segments may be attached only to pages of a file that are already allocated from the disk (and chained together). Extending (or contracting) a file must be done using other mechanisms (for example, see **SetEndOfFile** in the file package).

Swapping Strategies

A mechanism is provided for informing the segmentation machinery of emergency measures which can be taken when the signal **InsufficientVM** is (about to be) generated. These measures take the form of **SwappingProcedures** which, when called by the swapping manager, attempt to make more room in virtual memory and return a BOOLEAN indicating their success or failure to do so. The swapping manager invokes each procedure in turn, retrying the allocation after each procedure which has indicated success, until sufficient memory is obtained. If all such procedures indicate failure, the signal **InsufficientVM** is raised (the swapping manager is not crying wolf!).

The swapping strategies are maintained as a linked list of **SwapStrategy** nodes whose procedures are invoked from head to tail. The swapping manager initializes the list with a single node which invokes code swapping as a last resort.

```
SwappingProcedure: TYPE = PROCEDURE RETURNS [BOOLEAN];

SwapStrategy: TYPE = PRIVATE RECORD [
  link: POINTER TO SwapStrategy,
  proc: PUBLIC SwappingProcedure];

StrategyList: POINTER TO SwapStrategy ← @LastResort;
LastResort: SwapStrategy = SwapStrategy[NIL, TryCodeSwapping].
```

Swapping procedures are added to and removed from the list by the procedures:

AddSwapStrategy: PROCEDURE [strategy: POINTER TO SwapStrategy];

> The specified strategy node s is added to the head of the list of swapping procedures. If s is already on the list, its position and content are not disturbed.

RemoveSwapStrategy: PROCEDURE [strategy: POINTER TO SwapStrategy];

> The specified strategy node s is removed from the list of swapping procedures.

Currently, **TryCodeSwapping** uses round-robin to choose a code segment to swap out. Only code segments which are not locked are considered.

Since it is unattractive to require that swapping strategies (other than **TryCodeSwapping**) be locked, swapping procedures should observe the following conventions. If such a procedure obtains a state in which it has nothing to swap, it should either remove the node containing it from the strategy list or change the procedure in the node to be

```
CantSwap: SwappingProcedure = BEGIN RETURN [FALSE] END;
```

Because **CantSwap** is part of the swapping manager (and therefore locked), this will avoid swapping in a strategy procedure which knows it has nothing to do.

Miscellaneous Procedures

The following procedures implement conversion between memory addresses and virtual memory page numbers.

```
PageFromAddress: PROCEDURE [a: POINTER] RETURNS [PageNumber];

AddressFromPage: PROCEDURE [p: PageNumber] RETURNS [POINTER];

PagePointer: PROCEDURE [a: POINTER] RETURNS [POINTER];
```

PagePointer returns the address of the beginning of the page which contains its argument.

Alto/Mesa Storage Management Facilities

March 1, 1976

<MESA-DOC>STORAGE

Introduction

Two collections of Mesa procedures are available for acquiring and managing storage areas. The *segmentation machinery*, which is described in detail elsewhere, provides contiguous groups of *pages* (256 word blocks) in the virtual memory. A simplified interface with that machinery is described below. There is also a Mesa *free storage package* for managing arbitrarily sized *nodes* within free storage *zones*. Since all state information is recorded within the zones themselves, the system-provided instantiation of the latter package can manage an arbitrary number of zones. There is one system-defined zone, called the *heap*, available for general use, and special procedures exist for creating and destroying nodes within the heap. The salient characteristics of these packages are summarized below.

The segmentation machinery is most suitable for obtaining large blocks of storage. All bookkeeping information associated with such blocks is recorded in auxiliary tables that are managed by the segmentation system, not in the blocks themselves. Allocating or releasing a segment involves searching and updating a number of those tables and is relatively expensive. On the other hand, any freed page becomes available for general use by the system (loading, buffering, etc.) and any two adjacent free pages can be coalesced to become part of a new segment.

The free storage package is a transliteration of a BCPL program by Ed McCreight that was itself based upon a suggestion by Don Knuth (Volume 1, p. 453, #19). Within a zone, free nodes are kept as a linked list. One hidden word containing bookkeeping information is stored with each allocated node, and additional bookkeeping information is kept in the header of each zone. Allocation and release of nodes are usually very fast. Adjacent free nodes are always able to be coalesced. It is possible to add new areas of storage to enlarge a zone, but there are no procedures for shrinking a zone, even if an entire block becomes free.

The free storage package performs best when the sizes of nodes are small compared to the sizes of the block(s) making up the zone. In particular, the system's heap is intended to be used for small, transient data structures, such as the nodes of temporary list structure or the bodies of (short) strings when the maximum length must be computed dynamically or the structure must outlive the frame that creates it. Use of the heap for large (i.e., multipage) nodes decreases flexibility in storage management, since the additional pages become a permanent part of the heap.

The allocators in both packages return absolute pointers; allocated nodes are not relocatable and there is no garbage collection or automatic deallocation of any sort. Also, the values returned by the allocators are free pointers (type POINTER TO UNSPECIFIED) which must be cast appropriately (usually by assignment) before they can be used.

Segmentation Interface

The following definitions are contained in the file <MESA>SYSTEMDEFS.MESA.

AllocateSegment: PROCEDURE [nwords: INTEGER] RETURNS [base: POINTER]

> allocates a segment of virtual memory containing at least **nwords** words and returns the address of the first word in that segment. **AllocateSegment** provides a simple interface to NewSegment for allocating VM segments only; see the description of that procedure for further explanation.

SegmentSize: PROCEDURE [base: POINTER] RETURNS [nwords: INTEGER]

> returns the number of words actually obtained in the segment.

These two procedures allow complete utilization of segments obtained without knowledge of page structure and guaranteed only to have some minimum size. Such segments are returned to the system by

FreeSegment: PROCEDURE [base: POINTER] .

For programs in which the page structure is already known, the following procedures are also provided.

AllocatePages: PROCEDURE [npages: INTEGER] RETURNS [base: POINTER]

FreePages: PROCEDURE [base: POINTER];

PagesForWords: PROCEDURE [nwords: INTEGER] RETURNS [npages: INTEGER] .

Any storage obtained using **AllocatePages** is guaranteed to begin on a page boundary.

Free Storage Package

The following definitions are available in the file <MESA>FSPDEFS.MESA. A *zone* is a block of storage containing embedded *nodes.* The length of either a zone or a node is

BlockSize: TYPE = INTEGER [0..VMLimit/2] -- *15 bits.*

Each zone is headed by a **ZoneHeader**, which is a record with the following public fields:

```
. . .
threshold: BlockSize,  -- minimum node size in zone
checking: BOOLEAN, -- zone checking (see below)
. . .
```

Zones are identified by pointers of type

ZonePointer: TYPE = POINTER TO ZoneHeader .

An arbitrary block of (uninterpreted) storage is converted to a zone by

MakeZone: PROCEDURE [base: POINTER, length: BlockSize] RETURNS [z: ZonePointer];

such a block can alternatively be made an extension of an existing zone by

AddToZone: PROCEDURE [z: ZonePointer, base: POINTER, length: BlockSize] .

The largest node that can be allocated in a virgin block of size **length** is **nwords-ZoneOverhead.**

A node is allocated by

MakeNode: PROCEDURE [z: ZonePointer, n: BlockSize] RETURNS [POINTER] .

The value returned points to a block of **n** words; there is an additional hidden word of overhead (at offset -1) which must be preserved by users of the node. Nodes are sometimes split to satisfy allocation requests. Splitting within a zone **z** never generates fragments with size less than **z.threshold**, which is initialized to the minimum size of a free node. A request for a node of size **n** will produce a node with size in the range **[n . . n+z.threshold)**. The actual size of an allocated node is returned by

NodeSize: PROCEDURE [p: POINTER] RETURNS [BlockSize] .

If after coalescing all free nodes, a node of the requested size cannot be found,

NoRoomInZone: SIGNAL [z: ZonePointer]

is raised. This signal can be resumed (after, e.g., adding to the zone), and another attempt to allocate and return a suitable node will be made. An allocated node is returned to the zone by

FreeNode: PROCEDURE [z: ZonePointer, p: POINTER] .

Alternatively, an existing node can be split by calling

SplitNode: PROCEDURE [z: ZonePointer, p: POINTER, n: BlockSize];

the first **n** words of the node **p** remain allocated, and the remainder of the node is freed.

When a zone **z** is created, the variable **z.checking** is initialized to FALSE. If that variable is set to TRUE, the zone is checked for consistency prior to each transaction involving that zone. A failure raises one of the signals

InvalidZone: ERROR [POINTER];
InvalidNode: ERROR [POINTER] .

Allocation From The Heap

The following definitions are available in <MESA>SYSTEMDEFS.MESA. The heap is managed by the free storage package; the appropriate zone pointer for use with the procedures described in the previous section is returned by

HeapZone: PROCEDURE RETURNS [ZonePointer] .

The following procedures provide a specialized interface.

AllocateHeapNode: PROCEDURE [nwords: INTEGER] RETURNS [p: POINTER];
FreeHeapNode: PROCEDURE [p: POINTER] .

In addition,

AllocateHeapString: PROCEDURE [nchars: INTEGER] RETURNS [s: STRING]

allocates space for the body of a string in the heap. The field **s.length** is set to **0**; **s.maxlength**, to **nchars**. Such strings are freed by

FreeHeapString: PROCEDURE [s: STRING] .

If an allocation request cannot be satisfied from existing heap storage, an attempt is made to extend the heap with a block of appropriate size obtained from the segmentation machinery. The extension becomes a permanent part of the heap.

Alto/Mesa StreamIO Package

October 8, 1976

<MESA-DOC>STREAMIO

<MESA>STREAMIO.MESA contains a set of procedures for convenient use of the character and string stream facilities in Mesa. The procedures of the STREAMIO package are described below. The declarations necessary to use the procedures are in <MESA>IODEFS.MESA.

Initialization

The initial system provides an instance of STREAMIO which will obtain input from the keyboard and write output to the display. User programs may create new instances of STREAMIO to deal with other streams by writing

```
StreamIO: FROM "<mesa>streamio";
    . . .
DEFINITIONS FROM  . . . StreamIO . . .
    . . .
f: FrameHandle;
    . . .
f ← NEW StreamIO[InputStream, OutputStream];

Bind[f];  START f;
```

InputStream and **OutputStream** are StreamHandles for the desired input and output streams for the new instance of STREAMIO. The desired stream procedures may be accessed by OPENing **f**, writing **f.procedurename**, or by causing the appropriate procedure references to be bound to the new instance of STREAMIO. (When the control fault handler is implemented, binding the new instance of **StreamIO** will be performed automatically, eliminating the need for the explicit call on **Bind.**)

Character IO

ReadChar: PROCEDURE RETURNS [CHARACTER]

Returns the next character from the **InputStream.**

WriteChar: PROCEDURE [c: CHARACTER]

The CHARACTER c is written on the **OutputStream.**

String Input

The procedures below read input from the **InputStream.** The following exceptional conditions may occur.

LineOverflow: SIGNAL [STRING] RETURNS [STRING]

The input has filled the string, the current contents of the string is passed as a parameter to the SIGNAL. The catch phrase should return a string with more room.

Rubout: SIGNAL

The DEL key was typed during **ReadEditedString.**

The procedures are:

```
ReadEditedString: PROCEDURE [
  s: STRING,
  t: PROCEDURE [CHARACTER] RETURNS [BOOLEAN],
  newstring: BOOLEAN]
  RETURNS [CHARACTER]
```

s contains (on return) the string read from the **InputStream.** The procedure **t** returns TRUE if the CHARACTER passed to it should terminate the string. If (**newstring** is TRUE and the first input character is ESC) or (**newstring** is FALSE), then s is treated as if it had been read from **InputStream** (input characters are appended to it). Otherwise s is initialized to be empty.

A string is read from the **InputStream** with the following editing characters recognized:

↑A, ↑H (BS) delete the last character
↑W, ↑Q delete the last word
↑X delete the line and start over
↑R retype the line
↑V quote the next character

All characters except the terminating character are echoed on the **OutputStream.** The user supplied procedure **t** determines which character(s) terminate the string. The character returned is the character which terminated the string and is not echoed or included in the string.

The following procedures all call **ReadEditedString** passing TRUE for **newstring.**

```
ReadString: PROCEDURE [s: STRING,
  t: PROCEDURE [CHARACTER] RETURNS [BOOLEAN]]
```

Like **ReadEditedString** except that the terminating character is echoed. No value is returned.

```
ReadLine: PROCEDURE [s: STRING]
```

Reads from the **InputStream** up to the next carriage return character using **ReadEditedString.**

```
ReadID: PROCEDURE [s: STRING]
```

Uses **ReadEditedString** to read a string terminated with a space or carriage return into s. The terminating character is not echoed.

String Output

WriteString: PROCEDURE [s: STRING]

The string s is written on the **OutputStream.**

WriteLine: PROCEDURE [s: String]

The string s is written on the **OutputStream** followed by a carriage return.

Number Input

These procedures use the string-to-number conversion procedures from the STRINGS package.

ReadNumber: PROCEDURE [default: UNSPECIFIED, radix: CARDINAL]
RETURNS [UNSPECIFIED]

ReadID followed by **StringToNumber.** The value **default** will be displayed if esc is typed. **radix** is a default value, use the "B" or "D" notation to force octal or decimal. **radix** values other than 8 or 10 cause unpredictable results.

ReadDecimal: PROCEDURE RETURNS [INTEGER]

ReadID followed by **StringToDecimal.**

ReadOctal: PROCEDURE RETURNS [UNSPECIFIED]

ReadID followed by **StringToOctal.**

Number Output

NumberFormat: TYPE = RECORD [
base: [2..36], zerofill, unsigned: BOOLEAN, columns: [0..255]];

OutNumber: PROCEDURE [s: StreamHandle, val: UNSPECIFIED, f: NumberFormat]

val is converted to a character string on s in base **f.base.** The string is right justified in a field **f.columns** wide. If **f.zerofill,** the extra columns will be filled with zeros, otherwise spaces are used. If **f.unsigned,** the number is treated as unsigned.

WriteNumber: PROCEDURE [val: UNSPECIFIED, f: NumberFormat]

Equivalent to **OutNumber[OutputStream, val, f].**

WriteDecimal: PROCEDURE [n: INTEGER]

The value of **n** is converted to a character string of digits in base ten and output to the

OutputStream. Negative numbers are written with a preceeding minus sign ('-).

WriteOctal: PROCEDURE [n: UNSPECIFIED]

The value of **n** is converted to a character string of digits in base eight and output to the **OutputStream.** The numbers are unsigned, i.e., -2 is written as 177776B. The "B" is appended to any number more than one digit long.

Alto/Mesa Streams

December 1, 1976

<MESA-DOC>STREAMS

Introduction

The purpose of streams is to provide a standard interface between programs and their sources of sequential input and their sinks for sequential output. A set of standard operations defined for all types of streams is sufficient for all ordinary input-output requirements. In addition, most streams have special (device dependent) operations defined for them; programs which use such operations thereby forfeit complete compatibility.

Streams transmit information in atomic units called items. Usually an item is a CHARACTER or a WORD, and this is the case for most of the streams supplied with Mesa. Of course, a stream supplied to a program must have the same ideas about the kind of item it handles as the program does; otherwise confusion will result. Normally, streams which transmit text use CHARACTER items, and those which transmit binary information use WORDs.

Streams are passed about using StreamHandles, which are produced by the (device dependent) procedures that create streams (described in the sections below). A StreamHandle is a pointer to a variant record of type StreamObject, which is defined (in <MESA>STREAMDEFS.MESA) as follows:

```
StreamHandle: TYPE = POINTER TO StreamObject;

StreamObject: TYPE = RECORD [
  reset: PROCEDURE [StreamHandle],
  get: PROCEDURE [StreamHandle] RETURNS [UNSPECIFIED],
  putback: PROCEDURE [StreamHandle, UNSPECIFIED],
  put: PROCEDURE [StreamHandle, UNSPECIFIED],
  endof: PROCEDURE [StreamHandle] RETURNS [BOOLEAN],
  destroy: PROCEDURE [StreamHandle],
  body: PRIVATE SELECT PUBLIC type: * FROM
    Keyboard => . . .
    Display => . . .
    Disk => . . . ];
```

In addition, error conditions are reported in a fashion independent of the particular stream type, using the following definitions (not all error codes are applicable to all stream types):

```
StreamError: SIGNAL [stream:StreamHandle, error:StreamErrorCode];

StreamErrorCode: TYPE = {
  StreamType, StreamAccess, StreamOperation,
  StreamUnit, StreamPosition, StreamEnd, StreamBug};
```

As the definition implies, each stream object contains procedures that implement the standard stream operations, as described below (s is a StreamHandle, i is an item of the appropriate type,

and "code error" means that SIGNAL StreamError[s,code] is executed):

reset[s] restores the stream to some initial state, generally as close as possible to the state it is in just after it is created.

get[s] returns the next item; **StreamAccess** error if s cannot be read or if **endof[s]** is true before the call.

putback[s,i] modifies the stream so that the next **get[s]** will return i and leave s in the state it was in before the **putback.**

put[s,i] writes i into the stream as the next item; **StreamAccess** error if the stream cannot be written; **StreamEnd** error if there is no more space in the stream.

endof[s] TRUE if there are no more items to be gotten from s. For output streams, **endof** is device-dependent.

destroy[s] destroys s in an orderly way, freeing the space it occupies. Note that this has noting to do with deleting any underlying data structures or processes associated with the stream (like a disk file, for example, or the keyboard process).

Each of these operations is defined more precisely in the descriptions of the individual stream types which appear below. All of the stream routines produce the **StreamType** error when the variant of the **StreamObject** they are passes is not what they are expecting.

Keyboard Streams

The system comes equiped with a process that monitors the keyboard, keyset, and mouse buttons and buffers the input in one of a number of keyboard streams. The keyboard module (<MESA>KEYSTREAMS.MESA) provides a default keystream which is initially attached to the process. Interpretation of the standard stream operations is a follows:

reset[s] clears the buffer associated with s; any characters in the buffer are lost.

get[s] returns the next character in the buffer; if **endof[s]** is TRUE, busy-waits (BLOCKs if a scheduler is present in the system) until it is FALSE.

putback[s,i] modifies the stream so that the next **get[s]** will return i, independent of any type-ahead. If the buffer is full, **putback** is a no-op (sorry about that).

put[s,i] produces a **StreamAccess** error.

endof[s] TRUE if there are no characters in the buffer.

destroy[s] destroys s in an orderly way, freeing the space it occupies. Any characters in the buffer at the time of the **destroy** are lost. If s is the current keystream, the **StreamOperation** error results.

Initially, the default keystream is the current one; the keyboard process (see <MESA-DOC>PROCESS) always enters characters in the buffer of the current keystream, whose handle is returned by the procedure

GetCurrentKey: PROCEDURE RETURNS [StreamHandle];

Any number of keyboard streams may be created, using the procedure

CreateKeyStream: PROCEDURE RETURNS [StreamHandle];

A keystream is made current by calling the procedure

OpenKeyStream: PROCEDURE [stream:StreamHandle];

The stream which was current before the call is undisturbed, except that input characters are no longer directed to it by the keyboard process, but to stream instead. A keystream can be turned off (cease receiving characters) by calling

CloseKeyStream: PROCEDURE [stream:StreamHandle];

The StreamOperation error results if stream is not the current stream; otherwise the default keystream becomes current. The identity of the default keystream is returned by

GetDefaultKey: PROCEDURE RETURNS [StreamHandle];

Low-level keyboard functions: The keyboard process copies the current mouse coordinates into the cursor coordinates each times it runs (60 times per second). The basic system does not provide for lower level keyboard functions for for access to the keyset or mouse buttons through the stream. For the present programs wanting to read mouse and keyset buttons must read them directly from memory, either in an interrupt scheduled process or in some busy wait loop. For convenience the file KEYDEFS.MESA contains the declaration of a record containing all of the keyboard and mouse bits which may de directly mapped onto the keyboard words in memory.

```
updown: TYPE = {down, up};

KeyBits: TYPE = MACHINE DEPENDENT RECORD [
  blank: [0..377B],  -- not used
  Keyset1, Keyset2, Keyset3, Keyset4, Keyset5: updown,
  Red, Blue, Yellow: updown
  Five, Four, Six, E, Seven, D, U, V, Zero, K, Dash, P, Slash, BackSlash, LF, BS: updown,
  Three, Two, W, Q, S, A, Nine, I,
   X, O, L, Comma, Quote, RightBracket, TopSpare, MiddleSpare: updown,
  One, ESC, TAB, F, Ctrl, C, J, B,
   Z, LeftShift, Period, SemiColon, Return, Arrow, DEL, FL3: updown,
  R, T, G, Y, H, Eight, N, M,
   Lock, Space, LeftBracket, Equal, RightShift, BottomSpare, FL4, FL5: updown];

  Keys: POINTER TO KeyBits = -- magic memory location --;
```

Disk Streams

A disk stream is an array-like representation of a disk file; that is, parts of the file may reside in memory from time to time at the convenience of the stream. Like most arrays, a stream has a length; unlike array variables, the length of a stream may be changed by appending to it, and the maximum length is very large. Disk streams are created by the procedures

```
CreateByteStream, CreateWordStream: PROCEDURE [file:FileHandle, access:FileOptions]
  RETURNS [StreamHandle]
```

If **file** is a valid **FileHandle** with the appropriate access rights, it is opened and a byte or word stream is attached to it. If **access** is **Append** only, the stream is positioned at the end of the file, otherwise at the beginning. If **access** is null (zero), **Read** is assumed.

The stream's **FileHandle** and **FileOptions** may be read directly from the **StreamObject** using the field names **file** and **access**.

The operations allowed on the stream's length are determined by its access options (**FileOptions**); these options are negotiated with the underlying file system (see <MESA-DOC>SEGMENTATION). The options supported by the stream package are:

Read: the length is a constant.
Write: the length may decrease.
Append: the length may increase.

A disk stream has as part of its state a current **index** into the array representation of the file. The first data item is at index zero, the last at **length**-1. An invariant of a disk stream is **index** <= **length**. The current index and length are used in defining the semantics of the standard operations on disk streams, which are as follows:

reset: PROCEDURE [stream:StreamHandle]

Effect: sets the **index** to zero.

get: PROCEDURE [stream:StreamHandle] RETURNS [item:UNSPECIFIED]

If: **Read** IN **access** AND **index** < **length**.
Effect: **item** ← **stream**[**index**]; **index** ← **index**+1.

putback: PROCEDURE [stream:StreamHandle, item:UNSPECIFIED]

Effect: **StreamOperation** error.

put: PROCEDURE [stream:StreamHandle, item:UNSPECIFIED]

If: (**Write** IN **access** AND **index** < **length**) OR
(**Append** IN **access** AND **index** = **length**).
Effect: **stream**[**index**] ← **item**; **index** ← **index**+1;
length ← MAX[**index**, **length**].

endof: PROCEDURE [stream:StreamHandle] RETURNS [eof:BOOLEAN]

Effect: **eof** ← **index**=**length**.

destroy: PROCEDURE [stream:StreamHandle]

Effect: IF **Read** ~IN **access** AND **index**#0 THEN **length** ← **index** (i.e. truncate the file if it is not positioned at the beginning).

Actually, there is a little more to it. Disk streams deliver either byte or word items; in either case, the **index** is always computed in bytes. So the description above is a simplification of what really happens. Rather than clutter it up, suffice it to say that when accessing files in word mode, index values are always rounded up to word boundaries.

In addition to the standard operations, the following diskstream dependent functions are provided to efficiently copy large blocks of words to or from the stream:

```
ReadBlock: PROCEDURE [stream:StreamHandle, address:POINTER, words:INTEGER]
  RETURNS [count:INTEGER]

      If: Read IN access.
      Effect: count ← MIN[words,length-index];
        FOR index IN [index..index+count) DO
          MEMORY[address] ← stream[index];
          address ← address+1; ENDLOOP.

WriteBlock: PROCEDURE [stream:StreamHandle, address:POINTER, words:INTEGER]
  RETURNS [count:INTEGER]

      If: (Write IN access AND index < length) OR
        (Append IN access AND index = length).
      Effect: count ← IF Append IN access
          THEN words
          ELSE MIN[words,length-index];
        FOR index IN [index..index+count) DO
          stream[index] ← MEMORY[address];
          address ← address+1; ENDLOOP.
        length ← MAX[index, length].
```

When using ReadBlock and WriteBlock, the initial index must be on a word boundary (otherwise the StreamPosition error results). Note that the returned value may be less than words if the stream's access does not allow reading or writing of the whole block (the StreamAccess error is *never* raised by either of these procedures).

The stream index alluded to above is actually a structure:

```
StreamIndex: TYPE = RECORD [
  page: PageNumber,
  byte: WORD];
```

The first data byte of a stream is at StreamIndex[0,0]. The current stream position can be determined by calling

```
GetIndex: PROCEDURE [stream:StreamHandle] RETURNS [StreamIndex];
```

It is quite acceptable to do double-precision arithmetic on a StreamIndex (and even single precision operations on the individual fields, if you are carefull about borrows, carries, and overflows). The paged structure of the index can be restored by invoking

```
NormalizeIndex: PROCEDURE [index:StreamIndex] RETURNS [StreamIndex];
```

It returns an index whose byte field is in the range [0..CharsPerPage). The current index may be set by calling

```
SetIndex: PROCEDURE [stream:StreamHandle, index:StreamIndex];
```

Note that this may actually extend the file (with unspecified data) if Append access is allowed. To determine if this will happen, you might first want to call

```
FileLength: PROCEDURE [stream:StreamHandle] RETURNS [StreamIndex];
```

Note that FileLength returns the length as seen through the stream; this may differ from the

physical length of the disk file (if, for example, items have been appended to the stream but not yet written to the disk).

If a physical disk location is required along with the stream position, a file address (FA) will prove useful; it is similar to a StreamIndex with a disk address (DA) tacked on the front, except that the page field is one origin (in the Alto file system, page zero is the leader page).

```
FA: TYPE = MACHINE DEPENDENT RECORD [
  da: DA,
  page: PageNumber,
  char: WORD];
```

You may record the current stream position and re-establish it later, in a fashion similar to GetIndex and SetIndex, by calling the procedures

```
GetFA: PROCEDURE [stream:StreamHandle, fa:POINTER TO FA];

JumpToFA: PROCEDURE [stream:StreamHandle, fa:POINTER TO FA];
```

The special thing about JumpToFA is that the disk address in the fa is taken as a hint; if it doesn't work out (the page number or file serial number doesn't match the stream's version of them), JumpToFA will attempt to find the requested page via the shortest route and correct the fa accordingly. This may involve starting over at the beginning of the file. If that fails,

```
InvalidFP: SIGNAL [fp:POINTER TO FP];
```

will result, probably indicating that the file has been moved (or worse, deleted!) since the stream was attached to it. A call on some directory searching procedure may prove usefull in this situation, to determine if retrying the operation (with a new fp) is appropriate.

Alto/Mesa String Package

October 8, 1976

<MESA-DOC>STRINGS

<MESA>STRINGS.MESA contains procedures that implement various string operations. The necessary TYPE and PROCEDURE declarations appear in <MESA>STRINGDEFS.MESA and are described below.

TYPEs

```
SubStringDescriptor: TYPE = RECORD [
        base: STRING,
        offset, length: INTEGER]

SubString: POINTER TO SubStringDescriptor
```

SIGNALs

StringBoundsFault: SIGNAL [STRING]
An attempt was made to increase the length of the indicated string to be larger than the maxlength of the string.

Overflow: SIGNAL
An input number is too large. Decimal numbers must be in the range [-32767..32767]; octal numbers must be in the range [0..177777B].

InvalidNumber: SIGNAL
A string is not a valid number because it is empty or contains characters other than digits in the appropriate range.

PROCEDUREs

WordsForString: PROCEDURE [nchars: INTEGER] RETURNS [INTEGER]
Calculates the number of words of storage needed to hold a string of length **nchars**. The value returned includes any system overhead for string storage.

AppendChar: PROCEDURE [s: STRING, c: CHARACTER]
Appends the character **c** to the end of the string **s**.

AppendString: PROCEDURE [to, from: STRING]
Appends the string **from** to the end of the string **to**.

EqualString: PROCEDURE [s1, s2: STRING] RETURNS [BOOLEAN]
Returns TRUE if **s1** and **s2** contain exactly the same characters.

EquivalentString: PROCEDURE [s1, s2: STRING] RETURNS [BOOLEAN]
Returns TRUE if **s1** and **s2** contain the same characters except for case shifts. *Note that strings containing control characters may not be compared correctly.*

AppendSubString: PROCEDURE [to: STRING, from: SubString]
Appends the substring in **from** to the end of the string in **to.**

EqualSubString, EquivalentSubString:
PROCEDURE [s1, s2: SubString] RETURNS [BOOLEAN]
Analogous to EqualString and EquivalentString.

DeleteSubString: PROCEDURE [s: SubString]
Deletes the substring s from the string **s.base.**

The procedures below convert strings of ASCII characters to numbers in internal format.

StringToNumber: PROCEDURE [s: STRING, defradix: CARDINAL] RETURNS [INTEGER]
The characters of s are interpreted as a (possibly signed) number whose value is returned. **defradix** is used in the conversion unless the "B" or "D" notation is used. Supplying **defradix** values of other than 8 or 10 is not supported

StringToDecimal: PROCEDURE [s: STRING] RETURNS [INTEGER]
The characters of s are interpreted as a (possibly signed) decimal number whose value is returned.

StringToOctal: PROCEDURE [s: STRING] RETURNS [UNSPECIFIED]
The characters of s are interpreted as an octal number whose value is returned. The "B" notation of the source language is accepted but not required, *i.e.*, 1B2 = 100B = 100.

SECTION 2: MAXC DIRECTORIES FOR ALTO/MESA USERS

The Maxc directory <MESA> contains the source and object files of interest to the users of the Mesa system. The standard extension for Mesa source files is .MESA and the standard extension for Mesa object files is .XM. A Mesa object file contains three main parts: (1) object code, (2) a symbol table, and (3) source-to-object correspondence table (used by the debugger for source-level debugging).

The following list enumerates the files of interest to Mesa users. The object file corresponding to each source file listed below is also on <MESA>. The parenthesized file name following the name of DEFINITIONS modules is the name of the source file for the code which implements those definitions -- the ultimate documentation after all.

FSPDEFS (FSP) -- free-storage package
IODEFS (STREAMIO) -- some stream I/O
MENUDEFS (MENUS) -- command menus and selections
PROCESSDEFS (PROCESS) -- basic (hard) processes
RECTANGLEDEFS (RECTANGLES) -- display bitmaps
SEGMENTDEFS (SEGMENTS, FILES, SWAPPER) -- segmentation machinery
STREAMDEFS (STREAMS) -- disk stream package
(KEYSTREAMS, DISPLAY) -- keyboard and display
STRINGDEFS (STRINGS) -- string package
SYSTEMDEFS -- simplified interfaces to some facilities
WINDOWDEFS (WINDOWS) -- display windows

In addition to the source files listed above, all the source files for the debugger and other system components can be found on <MESA>. The <MESA> directory also contains MESA.IMAGE (the Alto/Mesa system), XDEBUG.IMAGE and XDEBUG.SYMBOLS (the Mesa "external" debugger), COMPILER.IMAGE (the compiler), MESA.RUN (the Alto/Bcpl program which "boots" the Mesa system on the Alto), and MESASYSTEM.CM (an FTP command file for transfering a complete Mesa environment from Maxc to your Alto disk). <MESA> also contains CHECKER.IMAGE (an error-correcting Mesa syntax checker). Section 3 has directions for running CHECKER.

The Maxc directory <MESA-DOC> contains the documentation for the Mesa system. Both .BRAVO and .EARS versions are maintained here. The following list enumerates the documentation files of interest:

MESASYSTEMCOVER -- the cover page
MESASYSTEM -- this material, excluding the subsections of Section 1
BIND -- binding facilities
DEBUGGER -- the debugger
DISPLAY -- display streams, windows, menus and selections
FILES -- file machinery
PROCESS -- low-level, hardware scheduled processes
STRINGS -- string manipulation package
STREAMIO -- character, string, and numerical I/O
STREAMS -- generalized stream access to I/O devices
SEGMENTS -- segmentation machinery

STORAGE -- simple segmentation interface, *heap*, and free-storage package
IMAGE -- Section 4: image files and image file facilities
MESASYSTEMDOCUMENT.EARS -- a compilation of all of the above files
MESA-NEWS.MSG -- an MSG-format news file

SECTION 3: HOW-TO-DO-IT

1. Finding an Alto capable of running Mesa

Any Alto with serial number 300 or greater should be capable of running Mesa. Altos with version 23 microcode and Alto II's also run Mesa.

2. Setting up your Alto disk

First, make sure your disk contains Alto Operating System version 5 or greater. Then obtain the text file <MESA>MESASYSTEM.CM from Maxc. This is a command file which will transfer the basic runtime files from Maxc to your Alto. You will need approximately 1000 free pages. This command file transfers: (1) MESA.RUN, a BCPL program which loads the ram with the Mesa emulator, loads main memory with the kernel Mesa system, and starts execution; (2) MESA.IMAGE, the core-image of the Mesa system; (3) XDEBUG.IMAGE, the "external" debugger; (4) XDEBUG.SYMBOLS, the symbols for the debugger; and (5) WMANLOADER.XM, the (experimental) window manager for use in the debugger. If the file MesaFont.Al exists, Mesa will use it for the system display; otherwise SysFont.Al is used.

The debugger must be installed (like Swat). Run the debugger by saying *mesa xdebug* to the Alto Executive. If you would like to include the window manager in the debugger, execute a NEW-BIND-START sequence on the file WMANLOADER.XM. Then execute the "↑Nstall" command (*control-N*). See the section on the debugger for more details.

3. Preparing your source file

Mesa accepts both unformatted ASCII and formatted BRAVO source text files. You will note that the Alto/Mesa debugger uses the source file to print source-text descriptions of the locus of the pc in frames and for setting breakpoints. In order to exploit this facility, you must be sure that the source file on your Alto disk is consistent with the object file.

4. Compiling your program

The Alto version of the compiler may be retrieved from <MESA>COMPILER.IMAGE. (Remember that compiled versions of all DEFINITIONS modules that your program uses must be on your disk.) Type *mesa compiler* to the Executive to invoke the compiler; it will prompt for the source file name. When it finishes, it will prompt again; a null filename will return you to the Alto Executive. Alternately, you may type *mesa compiler source1 source2* . . . directly to the Executive, making use of its filename completer if you wish.

Semantic errors result in a symbolic print-out of the location of the error (in the form: *procedure[character-position]*) and an indication of the type of error. The semantic passes try very hard to muddle through with a complete diagnosis. The compiler puts all error messages in the file *sourcename*.errlog.

Before using the compiler you may do a syntax check on the source. Obtain the binary

file <MESA>CHECKER.IMAGE and give the command *mesa checker* to the Executive. The checker will ask you for a source file and whether you wish to have the source scrolled on the screen while it is being checked. If an error occurs, the checker attempts to recover by deleting and/or inserting text (not in the file), displays the change(s), and asks whether you wish to plow on. Note that this is pure syntax analysis, no semantic (i.e. type) checking.

5. Running your program

And now the fun really begins ... Type *mesa* to the Alto Executive and, after a bit of disk rattling and ram loading, you will find yourself talking to the Mesa mini-debugger. At this point, you are well advised to browse through the debugger's documentation in Section 1. Basically, you must: (1) load your program and execute its initialization code -- NEW command, (2) bind the external references -- BIND command, and (3) start execution -- START command. When this fails, try putting in some breakpoints or enabling some tracing before executing step (3).

6. Talking to the debugger

In addition to the commands described in the section on the debugger, an experimental window manager allows you to view a number of display windows which contain the debugger's typescript, the user's typescript, and a current source file. Additional windows can be created to view other source files. The following sketchy description is meant to suffice until display windows are more fully integrated into the debugger.

It will be helpful if you try things out as you read the description below. Give the Executive the *mesa* command and then give the Debug command to enter the debugger. Be sure you have included WMANLOADER in the debugger when you installed it (see above).

The current window, which appears on top, is determined by the position of the cursor on the screen. The sensitive point of the cursor is generally the tip of the arrow, the center of the bullseye or the upperleft corner of the redbutton cursor. The left edge of the current window is a "jump bar". Moving into the jump bar gives you a double-arrow cursor to indicate that you are in scrolling mode: clicking red while any part of the cursor is in the jump bar will give you an uparrow which will allow you to scroll up the file, clicking green will give you a downarrow which will allow you to scroll down the file, and clicking yellow will give you a horizontal arrow which will allow you to to do absolute thumbing -- to the beginning if the arrow is at the top of the window, and to the end if it's at the bottom. A "thermometer" in the jump bar shows where you are. Moving out of the scrolling margin without clicking the cursor will allow you to return to selection mode.

When you're not in the jump bar, red selects a character and holding down the button allows you to extend the selection in either direction (much like Bravo, only the selection turns black); yellow selects a word and holding down the button allows you to extend by word boundaries in either direction (where words are considered to be a continuous stream of either spaces, controlchars, extra symbols, or alphanumerics). This is especially useful for selecting the names of files to be loaded into scratch windows or giving information to the debugger (more on this later). Holding down green will cause a menu of commands to appear to the left of the cursor. You may select a command by pointing at it (it will turn black); something will begin to happen when you release the button. If you change your mind prior to selecting a command, just move out of the menu before releasing green. If you decide not to go through with a selected command, simply click

green and that will resume the previous state as a default option. Currently, the commands are:

CREATE: Move the redbutton cursor around until you get to the place for the new window -- then click red. You will get a scratch window into which you can type things like filenames.

LOAD: The name of the file which will be loaded is the selection in the current window. Now move the redbutton cursor into any window (except a typescript window) and click red.

MOVE: The upper left-hand corner of the current window will stick to the redbutton cursor while you move it around. Click red to unstick it, click green to return to the former position.

GROW: The current window will turn gray and the lower right-hand corner of the window will stick to the redbutton cursor while you change the size (subject to a minimum size limitation). Click red to unstick it, click green to return to its former size.

DESTROY: Move the bullseye cursor into the window you want to kill off and click red. If you try to destroy any of the typescript windows, it will ignore you .

STUFF IT: Takes the current selection of the active window and stuffs it into the input stream of the window that is selected by clicking red.

Examples of the use of STUFF IT:

This command is useful for selecting procedure or program names as well as a unique string for setting a breakpoint. It might be helpful to create a scratch window that contains frequently used procedure names, program names, Debugger command letters, and even a space and carriage return (for lazy typists). Then you might select the identifier of your choice and "stuff it" into the debugger's window eliminating the hazards and time lost by incorrect typing and searching. You can also go into the source file of a program and select a unique word or string of words that the debugger can then use to set a breakpoint.

After any action is taken, the current window is refreshed and its selection is updated.

Good luck! Be of good humor! And please, please feel free to send suggestions, requests, plaudits, and complaints to Chuck Geschke who will darken or brighten the day of the Mesan to whom they should be directed.

SECTION 4: IMAGE FILES

<MESA-DOC>IMAGE

December 1, 1976

A Mesa image file contains the code, data, and control information necessary to start execution of a Mesa system. This section defines the format of image files and the facilities provided to make them.

Format:

```
ImageHeader: TYPE = MACHINE DEPENDENT RECORD [
  version: CARDINAL,  -- should be 1
  options: WORD,  -- should be 0
  av, gft, sd: POINTER,
  state: StateVector,
  map: ARRAY OF MapItem];

MapItem: TYPE = MACHINE DEPENDENT RECORD [
  page, count: [0..255]];
```

The first data page of an image file is a record of type **ImageHeader**. The size of the array depends on the number of page groups in the file. The last element of the array is **MapItem[0,0]**. The **version** field identifies the version of the format being used (currently only version 1 is defined). The **options** field specifies other data about the image file as a bit mask. Currently no options are supported. The three pointers, **av**, **gft**, and **sd** are the initial values for those processor registers. The StateVector in **state** is the initial state of the program. This StateVector will be loaded as the lowest priority process and will be started using the Mesa TRANSFER WITH construct.

After the first page, the remaining pages of the file contain the pages to be loaded into memory. The entries in the **map** array identify the core locations and number of pages in each page group. The MESA.RUN program which loads an image file will load page groups until it encounters **MapItem[0,0]**.

Making Image Files:

The basic system contains code to make an image file of itself and any user programs which have been loaded. Users may invoke this code either with the mini debugger's MakeImage command or by calling the following procedure:

```
MakeImage: PROCEDURE [name: STRING, symbolsToImage: BOOLEAN];
```

If **symbolsToImage** is TRUE then all symbol table segments in the system will be copied into the image file. Since symbol tables are usually very large (compared to code) using this option will create very large image files. In particular, the size of the image file will be approximately the sum of the sizes of MESA.IMAGE and any user program files included. When **symbolsToImage** is FALSE the symbol segments are left attached to their original files. The **MakeImage** procedure

contains code to lookup these files and reattach the symbol tables when the image file is restarted. If any symbol file is not present, the system behaves as if the symbol segment had been deleted. There is no check to assure that the files found when the image file restarts contain the same data as the files of the same names when the image file was created.

The mini debugger's MakeImage command accepts a filename, defaults the extension to ".image" and calls MakeImage[name,FALSE]. The MakeImage procedure always returns to the Alto Executive since there is no graceful way to continue from making an image file. When the image file is restarted, MakeImage returns to its caller as if nothing had happened. If MakeImage was called by the mini debugger, the mini debugger will be ready to accept a new command. A program which calls MakeImage may continue executing normally.

Restrictions:

1. The name of the new image file may not be the same as the name of the image file running at the time MakeImage is called. An image file can be renamed any time it is not running.

2. The user program should not have handles on any files or disk streams. Any FileSegments allocated by a user program will be made a part of the new image file.

3. MakeImage should only be called from process priority level 14, i.e. the process at which the mini debugger starts running.

4. This list of restrictions may not be exhaustive. In general you should avoid doing anything other than loading and binding modules and initializing data structures before making an image file.

<koalkin>debugsum.bravo

DEBUGGER SUMMARY

DEBUGGER COMMANDS:

BInd
BReak Entry
Exit
CAse Ignore
CAse Heed
CLear All Breaks [confirm]
Entries [confirm]
Traces [confirm]
Xits [confirm]
Break
Entry Break
Trace
Program Break
Trace
Trace
Xit Break
Trace
COremap [confirm]
CReate
Display Binding path
Eval-stack
Frame
Module
Stack
Variable
↑Nstall [confirm]
Interpret Array
Call
De-reference
Expression
Pointer
Size
@
Load
New
Octal Clear Break
Set Break
Read
Write
Proceed [confirm]
Quit [confirm]
Reset context
SEt Break
Context
Octal Context
Program Break
Trace
Trace
STart
Trace All Entries
Exits
Entry
Exit

WHAT DEBUGGER MOUSE BUTTONS DO:

	ScrollBar	TextArea
RED	ScrollUp	Select/Extend characters
YELLOW	Thumb	Select/Extend words
GREEN	ScrollDown	Menu Commands

MENU COMMANDS:

CREATE
DESTROY
MOVE
GROW
LOAD
STUFF IT

WHAT MENU MOUSE BUTTONS DO:

RED	"Do it" - in this window/ at this spot
GREEN	Reset to previous state

DURING TYPE IN TO TYPESCRIPT WINDOW:

↑A	delete character	↑X	delete line
↑H	delete character	↑R	retype line
↑W	delete word	↑V	" next
↑Q	delete word	esc	old string

Programmer's Addendum:
Mesa Language Manual, Version 1.0

The initial release of the Mesa Language Manual is intended to present the Mesa language as it will be when the compiler is transferred from PARC to ITG/SDD in the first quarter of 1977. This addendum describes the differences between the current language and the language as given in the manual. It also corrects any errors which have been detected in the Manual. Lastly, it includes a reference grammar for the current language (which will eventually be included as an appendix to the manual itself).

Mesa users are urged to report any discrepancies which they find in the manual or in this addendum so they can be corrected in future versions. In future versions of the addendum we will give credit to the first person to report a given error. Any person who finds more than one distinct, previously unreported error in a single release of the manual or the addendum will also receive a commendation suitable for framing.

Jim Mitchell, 2 Nov 76

A.1 Errata in Version 1.0 of the Mesa Language Manual

This section is a partial reproduction of the table of contents for the manual with errata inserted following the section heading for the section in which they appear. This is an experiment, and I would appreciate any feedback on the utility of listing errors this way or any alternate ways of doing so.

The compiler currently does not do range checking on the use of subrange variables at either compile or run time.

Empty intervals are not allowed in declarations. However, they can be used when specifying the bounds for a FOR loop or with the IN relational operator.

On the Alto, if a subrange covers more than 2^{15}-1 values, its internal representation (which always represents the lowest value in the range as zero) will include some values which, viewed as integers, are negative. Thus, unsigned arithmetic will be needed for adding, subtracting, comparing, etc. values in ranges. At the moment, this is not guaranteed to work correctly, except for the fundamental operators (←, =, and #).

SETS ARE NOT YET IMPLEMENTED AT ALL. Enumerated types do work, however.

General expressions cannot be used when indexing or when calling a procedure; in particular, the form (exp)[exp] is not allowed.

Default values cannot be specified for components of records. Thus, any components defaulted in a constructor (sec. 3.4.4) have an indefined value.

The current compiler does not pack components in records in any optimal way. It only guarantees that fields which require less than one word do not overlap a word boundary.

An extractor may not contain an imbedded extractor: i.e., an assignment such as the following is not allowed:

[a, [b, c, d], e] ← someRecordValue;

ORDERED pointer types are not implemented.

Most automatic dereferencing is not implemented except for single-level dereferencing used with pointer qualification. For example, *ptr.component* and *ptr↑.component* are equivalent, but if *ptr2* were defined as follows:

ptr2: POINTER TO POINTER TO *foo;*

then *ptr2.component* is *not* equivalent to *ptr2↑↑.component.* In fact, *ptr2.component* is not type-correct and will cause an error. Automatic dereferencing without qualification is not implemented at all, thus the example in this section

candidate1 ← winner;

will cause an error from the compiler.

A.2 Collected Grammar for Mesa

The Mesa grammar in this section has been divided into four parts, corresponding to the syntax for **CompilationUnit**, **TypeSpecification**, **Statement**, and **Expression**. These four parts refer to each other and occasionally use syntax rules from other parts (such as **LeftSide** which is used in an assignment statement but defined under **Expression**). Where such cross references occur, a comment has been added to indicate which part to refer to. Other than this, each part is self-contained, and the productions within a part have been ordered alphabetically by their names -- except that the productions for **CompilationUnit**, **TypeSpecification**, etc. each head their respective sections. Feedback on this ordering or on alternatives would be appreciated.

CompilationUnit ::=

```
                    Directory
                    DefinitionsFrom
                    identifier : ModuleHead = GlobalAccess
                    ModuleBody
Declaration     ::= IdList : Attribute TypeSpecification Initialization ; |
                    IdList : Attribute TYPE = Attribute TypeSpecification ; |
                    IdList : External TypeSpecification ;
DeclarationSeries   ::= empty | DeclarationSeries Declaration
Directory       ::= empty | DIRECTORY IncludeList ;
DefinitionsFrom ::= empty | DEFINITIONS FROM IdList ;
FileName        ::= stringLiteral
GlobalAccess    ::= Access                                  -- in Declaration
IdList          ::= identifier | IdList , identifier
Implementing    ::= empty | IMPLEMENTING identifier
IncludeList     ::= identifier : FROM FileName |
                    IncludeList , identifier : FROM FileName
ModuleBody      ::= BEGIN
                    OpenClause                              -- in Statement
                    DeclarationSeries
                    StatementSeries
                    END .
ModuleHead      ::= PROGRAM ModuleParams Implementing Sharing |
                    DATA    ModuleParams Implementing Sharing |
                    DEFINITIONS Sharing
ModuleParams    ::= empty | [ NamedFieldList ]              -- in Declaration
Sharing         ::= empty | SHARING IdList
StatementSeries ::= empty | Statement |
                    StatementSeries ; Statement
```

TypeSpecification ::=

```
                    TypeIdentifier |
                    TypeConstructor
Access          ::= empty | PUBLIC | PRIVATE
Adjective       ::= identifier
ArrayDescriptorTC   ::= DESCRIPTOR FOR ArrayTC
ArrayTC         ::= ARRAY IndexType OF TypeSpecification |
                    ARRAY OF TypeSpecification
Attribute       ::= Access Protection
CommonPart      ::= empty | NamedFieldList ,
EnumerationTC   ::= { IdList }
External        ::=  empty | EXTERNAL
FieldList       ::= [ UnnamedFieldList ] | [ NamedFieldList ]
IndexType       ::= SubrangeTC | EnumerationTC | TypeIdentifier
Initialization  ::= empty | ← InitExpr | = InitExpr
```

```
InitExpr          ::= Expression |
                      ProcedureLiteral |
                      [ Expression ] |
                      CODE
Interval          ::= [ Expression .. Expression ] |
                      [ Expression .. Expression ) |
                      ( Expression .. Expression ] |
                      ( Expression .. Expression )
MachineDependent  ::= empty | MACHINE DEPENDENT
NamedFieldList    ::= IdList : Attribute TypeSpecification |
                      NamedFieldList , IdList : Attribute TypeSpecification
ParameterList     ::= empty | FieldList
PointerTC         ::= POINTER TO Protection TypeSpecification
PortTC            ::= PORT ParameterList ReturnsClause
PredefinedType    ::= INTEGER | BOOLEAN | CARDINAL |
                      CHARACTER | STRING
ProcedureLiteral  ::= BEGIN
                      OpenClause                    -- in Statement
                      DeclarationSeries             -- in CompilationUnit
                      StatementSeries               -- in CompilationUnit
                      END
ProcedureTC       ::= PROCEDURE ParameterList ReturnsClause
Protection        ::= empty | READ-ONLY
RecordTC          ::= MachineDependent RECORD [ VariantFieldList ]
ReturnsClause     ::= empty | RETURNS FieldList
SignalOrError     ::= SIGNAL | ERROR
SignalTC          ::= SignalOrError ParameterList ReturnsClause
SubrangeTC        ::= Interval | TypeIdentifier Interval
Tag               ::= identifier : Attribute TagType |
                      COMPUTED TagType
TagType           ::= TypeSpecification | *
TypeConstructor   ::= ArrayDescriptorTC | ArrayTC | EnumerationTC |
                      PointerTC | PortTC | ProcedureTC | RecordTC |
                      SignalTC | SubrangeTC
TypeIdentifier    ::= PredefinedType | identifier |
                      identifier . identifier |
                      Adjective TypeIdentifier
UnnamedFieldList  ::= TypeSpecification |
                      UnnamedFieldList , TypeSpecification
Variant           ::= IdList => [ VariantFieldList ] , |
                      IdList => NULL ,
VariantFieldList  ::= CommonPart identifier : Attribute VariantPart |
                      VariantPart |
                      NamedFieldList |
                      UnnamedFieldList
VariantList       ::= Variant | VariantList Variant
VariantPart       ::= SELECT Tag FROM
                      VariantList
                      ENDCASE
```

Statement ::=

AssignmentStmt | BindStmt | Call |
CompoundStmt | ContinueStmt | ErrorCall |
ExitStmt | GotoStmt | IfStmt | LoopStmt |
NullStmt | ResumeStmt | RetryStmt |
ReturnStmt | SelectStmt | SignalCall |
StartStmt | StopStmt

AdjectiveList ::= Adjective | AdjectiveList , Adjective -- in TypeSpecification

Assignation ::= FOR identifier ← Expression , Expression

AssignmentStmt ::= LeftSide ← Expression | -- LeftSide in Expression
Extractor ← Expression

BindStmt ::= BIND Expression

Body ::= OpenClause
EnableClause
StatementSeries -- in CompilationUnit

Call ::= LeftSide | -- in Expression
LeftSide [ComponentList] | -- ComponentList in Expression
[ComponentList ! CatchPhrase]

Catch ::= IdList => Statement | ANY => Statement

CatchPhrase ::= Catch | CatchSeries ; Catch

CatchSeries ::= IdList => Statement |
CatchSeries ; IdList => Statement

ChoiceSeries ::= AdjectiveList => Statement ; |
ChoiceSeries AdjectiveList => Statement ;

CompoundStmt ::= BEGIN
Body
ExitsClause
END

ConditionTest ::= empty | WHILE Expression | UNTIL Expression

ContinueStmt ::= CONTINUE

Direction ::= empty | INCREASING | DECREASING

ElseClause ::= empty | ELSE Statement

EnableClause ::= ENABLE Catch ; |
ENABLE BEGIN CatchSeries END ;

ErrorCall ::= ERROR Call | ERROR

ExitsClause ::= EXIT ExitSeries | EXIT ExitSeries ;

ExitSeries ::= LabelList => Statement |
ExitSeries ; LabelList => Statement

ExitStmt ::= EXIT

ExtractItem ::= empty | LeftSide

Extractor ::= [KeywordExtractList] |
[PositionalExtractList]

FinalStmtChoice ::= empty | => Statement

FinishedExit ::= FINISHED => Statement |
FINISHED => Statement ;

GotoStmt ::= GOTO Label | GO TO Label

IfStmt ::= IF Expression THEN Statement ElseClause

Iteration ::= FOR identifier Direction IN Subrange -- Subrange in Expression

IterativeControl ::= empty | Repetition | Iteration | Assignation

KeywordExtract ::= identifier : ExtractItem

KeywordExtractList ::= KeywordExtract |
KeywordExtractList , KeywordExtract

Label ::= identifier

LabelList ::= Label | LabelList , Label

LeftItem ::= Expression

LoopControl ::= IterativeControl ConditionTest

LoopExits ::= ExitSeries | FinishedExit | ExitSeries ; FinishedExit

```
LoopExitsClause ::= empty | REPEAT LoopExits
LoopStmt        ::= LoopControl
                    DO
                    Body
                    LoopExitsClause
                    ENDLOOP
NullStmt        ::= NULL
OpenClause      ::= empty | OPEN OpenList ;
OpenItem        ::= Expression | identifier : Expression
OpenList        ::= OpenItem | OpenList , OpenItem
PositionalExtractList ::= ExtractItem |
                          PositionalExtractList , ExtractItem
Repetition      ::= THROUGH Subrange                    -- in Expression
ResumeStmt      ::= RESUME |
                    RESUME [ ComponentList ]            -- ComponentList in Expression
RetryStmt       ::= RETRY
ReturnStmt      ::= RETURN |
                    RETURN [ ComponentList ]            -- ComponentList in Expression
SelectStmt      ::= Select | SelectVariant
Select          ::= SELECT LeftItem FROM
                        StmtChoiceSeries
                        ENDCASE FinalStmtChoice
SelectVariant   ::= WITH OpenItem SELECT TagItem FROM
                        ChoiceSeries
                        ENDCASE FinalStmtChoice
SignalCall      ::= SIGNAL Call
StartStmt       ::= START LeftSide
StmtChoiceSeries    ::= TestList => Statement ; |
                        StmtChoiceSeries TestList => Statement ;
StopStmt        ::= STOP | STOP [ ! CatchPhrase ]
TagItem         ::= empty | Expression
Test            ::= Expression | RelationTail           --RelationTail in Expression
TestList        ::= Test | TestList , Test
```

Expression ::=

```
                    AssignmentExpr | Disjunction | IfExpr |
                    NewExpr | SelectExpr | SignalCall
AddingOp        ::= + | -
AssignmentExpr  ::= LeftSide ← Expression
ChoiceList      ::= AdjectiveList => Expression , |      -- AdjectiveList in Statement
                    ChoiceList AdjectiveList => Expression ,
Component       ::= empty | Expression
ComponentList   ::= KeywordComponentList | PositionalComponentList
Conjunction     ::= Negation | Conjunction AND Negation
Constructor     ::= OptionalTypeId [ ComponentList ]
Disjunction     ::= Conjunction | Disjunction OR Conjunction
ExprChoiceList  ::= TestList => Expression , |
                    ExprChoiceList TestList => Expression ,
ExpressionList  ::= Expression | ExpressionList , Expression
Factor          ::= - Primary | Primary
IfExpr          ::= IF Expression THEN Expression ELSE Expression
IndexedReference    ::= LeftSide [ Expression ]
KeywordComponent ::= identifier : Component
KeywordComponentList ::=  KeywordComponent |
                          KeywordComponentList , KeywordComponent
```

```
LeftSide            ::= identifier |
                        Call | IndexedReference |
                        ( Expression ) . identifier | LeftSide . identifier |
                        ( Expression ) ↑          | LeftSide ↑ |
                        COERCE [ Expression ] |
                        COERCE [ Expression , TypeIdentifier ]
Literal             ::= numericLiteral |      -- all defined outside the grammar
                        stringLiteral |
                        characterLiteral
MultiplyingOp       ::= * | / | MOD
Negation            ::= Relation | Not Relation
NewExpr             ::= NEW LeftSide
Not                 ::= ~ | NOT
OptionalTypeId      ::= empty | TypeIdentifier
PositionalComponentList ::=  Component |
                             PositionalComponentList , Component
Primary             ::= LeftSide | Literal | Constructor |
                        ( Expression ) | @ LeftSide |
                        LENGTH [ LeftSide ] | BASE [ LeftSide ] |
                        MIN [ ExpressionList ] | MAX [ ExpressionList ] |
                        ABS [ Expression ] |
                        TypeOp [ TypeIdentifier ] |
                        DESCRIPTOR [ Expression ] |
                        DESCRIPTOR [ Expression , Expression ] |
                        DESCRIPTOR [ Expression , Expression , TypeIdentifier ]
Product             ::= Factor | Product MultiplyingOp Factor
Relation            ::= Sum | Sum RelationTail
RelationalOp        ::= # | = | < | <= | > | >=
RelationTail        ::= RelationalOp Sum | Not RelationalOp Sum |
                        IN SubRange | Not IN Subrange
SelectExpr          ::= SelectExprSimple | SelectExprVariant
SelectExprSimple    ::= SELECT LeftItem FROM
                             ExprChoiceList
                             ENDCASE => Expression
SelectExprVariant   ::= WITH OpenItem SELECT TagItem FROM
                             ChoiceList
                             ENDCASE => Expression
Subrange            ::= SubrangeTC | TypeIdentifier
Sum                 ::= Product | Sum AddingOp Product
TypeOp              ::= SIZE | FIRST | LAST
```

A.3 Interim JOIN procedure

The JOIN procedure below is self-contained and needs no DEFINITIONS module to be included in any program which contains a copy of it. A program which would have used a JOIN statement of the form

JOIN progA.portA TO progB.portB

should instead contain a copy of the JOIN procedure (available as a separate Bravo file on Maxc as <MITCHELL>JOINPROC.MESA) and call it as

JOIN[@progA.portA, @progB.portB]

Actually, this procedure also simultaneously accomplishes the effect of the statement

JOIN progB.portB TO progA.portA

```
JOIN: PROCEDURE [p1, p2: POINTER] =
    BEGIN
    Port: TYPE = MACHINE DEPENDENT RECORD
        [
        framePtr: POINTER,   -- actually a pointer to a frame
        link: POINTER TO Port
        ];
    port1.link ← p2;              -- JOIN 1 TO 2
    port2.link ← p1;              -- JOIN 2 TO 1
    END;
```